UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ROBERTA PASQUALLI

IDENTIFICAÇÃO E DIAGNÓSTICO DE ERROS DE MODELAGEM CONCEITUAL COMETIDOS POR APRENDIZES

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Raul Sidnei Wazlawick
(Orientador)

Florianópolis, junho de 2002

IDENTIFICAÇÃO E DIAGNÓSTICO DE ERROS DE MODELAGEM CONCEITUAL COMETIDOS POR APRENDIZES

ROBERTA PASQUALLI

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração em Sistemas de Conhecimento e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Fernando Álvaro Ostuni Gauthier, Dr.
(Coordenador do Curso)

Banca Examinadora

Prof. Raul Sidnei Wazlawick, Dr.
(Orientador)

Prof. Vitório Bruno Mazzola, Dr.

Profª. Patrícia Vilain, Dr.

Prof. Ricardo Pereira e Silva, Dr.

"Não basta ensinar ao homem uma especialidade,
porque se tornará assim uma máquina utilizável,
mas não uma personalidade. É necessário que
adquira um sentimento, um senso prático daquilo que
vale a pena ser empreendido, daquilo que é belo,
do que é moralmente correto. A não ser assim,
ele se assemelhará com seus conhecimentos profissionais,
mais a um cão ensinado do que a uma
criatura harmoniosamente desenvolvida.
Deve aprender a compreender as motivações dos homens,
suas quimeras e suas angústias, para determinar com exatidão
seu lugar preciso em relação a seus próximos e à comunidade"

Albert Einstein

Este trabalho é para ***Deus***,
Ser Superior, que tudo vê e a ninguém
desampara.
Para ***mamãe***, por entender que chega
uma hora na vida em que é preciso largar tudo
e partir.
À ***Marcio Antonio Bianchi***, por acalmar meu coração.
Para ***Eliete Marchioro***, amiga do coração, desabafo
nas horas tristes
e sorrisos nas boas horas.
Josimar de Aparecido Vieira, por
acreditar que eu sou capaz.
E, recordando meu orientador, ***Raul Sidnei***
Wazlawick lembro de Isaac Newton dizendo que
se avistei mais longe é porque
estive em ombros de gigantes.

A ***Heitor*** e ***Tereza***, pai e mãe, justos, honestos e corretos,
que durante tantos anos, com
com seus sábios ensinamentos, fundamentaram-me
na busca da verdade.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ANEXOS

LISTA DE SIGLAS

BNF – Backus-Naur Form
CASE – Engenharia de Software Ajudada por Computador
LPOO – Linguagem de Programação Orientada a Objetos
OO – Orientação a Objetos
OMG – Object Management Group
OMT – Object Modeling Technique
UFSC – Universidade Federal de Santa Catarina
UML – Unified Modeling Language

RESUMO

Este trabalho apresenta uma abordagem para a identificação e para o tratamento dos erros realizados por aprendizes no desenvolvimento de modelagem conceitual. Para fundamentar teoricamente este trabalho foi realizado um estudo sobre Orientação a Objetos, UML (Unified Language Modeling) e, principalmente sobre quais são os erros que mais comumente são observados quando os aprendizes desenvolvem seus próprios modelos conceituais. Além de procurar por erros comuns relatados na bibliografia, procurou-se analisar trabalhos de alunos e catalogar os erros observados, a fim de verificar se é possível automatizar a detecção dos mesmos. Este tema foi escolhido, pois a partir desta identificação e catalogação de erros, abre-se a possibilidade de aprimoramento das ferramentas CASE existentes e até mesmo o desenvolvimento de novas ferramentas que sugiram meios de auxiliar o aluno no momento de desenvolvimento da modelagem conceitual. Foram identificados ao todo 14 (quatorze) tipos de erros de modelagem, dos quais 06 (seis) são facilmente diagnosticados através de uma simples análise sintática do modelo, outros erros já necessitam de um tratamento semântico mais profundo para serem identificados e tratados.

ABSTRACT

This work presents a proposal for the identification and for the treatment of mistakes made by learners in the development of conceptual modeling. To justify theoretically this work, a study about Orientation to Objects, UML (Unified Language Modeling) was accomplished, mainly about the kind of errors that are more commonly observed when students develop their own conceptual models. Besides looking for common mistakes related in the bibliography, we tried to analyze students works and listed the observed mistakes, with the intention to verify if it is possible to make their detection on an automatic way. This subject was elected because after the identification and listing of the mistakes, there may be the opening of the possibility of improvement of the CASE tools existent and also the development of new tools that may suggest ways of helping students at the moment of conceptual modeling. Fourteen kinds of modeling errors were identified, from which six can be easily detected through a simple syntactic analysis on the model. Other mistakes need a deeper semantic treatment in order to be identified.

CAPÍTULO I

ABORDAGEM GERAL

1.1. Aspectos Gerais do Trabalho

Atualmente as metodologias influenciadas pelo paradigma de orientação a objetos têm recebido um maior destaque justamente por possibilitarem, de forma mais adequada, o tratamento de sistemas complexos.

Segundo BUZATO (1998), a modelagem orientada a objetos tem como objetivo representar o mundo real através de objetos. Esses objetos podem ser de vários tipos, como por exemplo, entidades físicas (aviões, robôs, etc) ou abstratas (vendas, pagamentos, etc). A característica mais importante da abordagem orientada a objetos para desenvolvimento de software é a unificação, através do conceito de objeto, de dois elementos que tradicionalmente têm sido considerados separadamente em paradigmas de programação mais antigos: dados e funções.

Um estudo mais aprofundado sobre este aspecto está descrito no capítulo II deste trabalho.

As linguagens de modelagem têm sido propostas como uma solução para a documentação de cada estágio de um projeto de software. Verificou-se após várias propostas metodológicas, que o resultado de cada fase pode ser preservado e utilizado na fase seguinte.

A modelagem orientada a objetos é vista, cada vez mais, como a melhor opção para modelagem de sistemas, e a UML (Unified Modeling Language), devido ao fato de ser uma tentativa de unificar todas as linguagens existentes para a especificação de sistemas orientados a objetos, tem grandes chances de se tornar um padrão de fato, pois foi proposta por três expoentes da área Booch, Rumbaugh e Jacobson.

A notação da UML, que é padrão OMG (Object Management Group), é uma combinação de sintaxes vindas de várias fontes, com vários símbolos oriundos das técnicas existentes e com alguns símbolos novos agregados para tratar situações específicas.

Associados à notação UML existem fases do processo de análise e projeto proposto por (LARMAN, 2002) que podem ser definidas como Diagrama de Casos de Uso, Modelagem Conceitual, Diagramas de Seqüência do Sistema, Diagramas de Colaboração, Atribuição de Responsabilidade, Diagramas de Classes. Outros autores restringem a apenas alguns desses passos, com é o caso de FURLAN (1998) que cita apenas os Diagramas de Classes, Diagramas de Casos do Uso, Diagrama de Interação e Diagrama de Estados, os quais descrevem uma possível ordem das atividades e um ciclo de vida de desenvolvimento. A UML, por ser parte central deste trabalho estará abordada com maior precisão no capítulo III.

Uma das fases em várias propostas de ciclo de desenvolvimento é a fase onde o modelo conceitual é definido. Nesta fase, é comum ao aprendiz cometer erros que conseqüentemente causarão falhas na modelagem final, provocando problemas ao programador do sistema.

Neste sentido, torna-se importante que, no momento da aprendizagem, sejam detectados os erros de modelagem cometidos pelo aprendiz, porque na vida profissional tais erros não serão tolerados. Por ser este o enfoque principal do trabalho, o capítulo IV fará referências abrangentes sobre a modelagem conceitual.

O capítulo V detalhará os erros mais comuns realizados pelos acadêmicos de graduação e pós-graduação das disciplinas de Análise de Sistemas e Orientação a Objetos.

Conhecendo quais são os erros mais freqüentemente encontrados, surge a necessidade de catalogar estes erros, para que possam ser analisados, podendo-se, desta forma identificar qual é o melhor modo de tratá-los.

Para isso, foram criadas operações que definem o domínio do problema que será utilizado, conforme a notação BNF Estendida (Backus-Naur Form) e, a partir dessa formalização serão definidos os algoritmos (ANEXO 01) necessários para que a

informação ao aprendiz de desenvolvimento da modelagem conceitual da UML possa ser aplicada de maneira correta, algoritmos estes que darão suporte a validação do modelo proposto e que podem vir a dar suporte para trabalhos futuros de implementação em forma de software da proposta. A formalização dos dados, juntamente com a implementação do modelo proposto para validação será abordados no capítulo VI.

Para a validação do modelo proposto, além da sintaxe definida para a formulação dos algoritmos (ANEXO 01), foram escolhidos de forma esporádica 2 (dois) erros mais freqüentemente cometidos por aprendizes, que foram implementados na Linguagem de Programação C, versão BC31 da Borland. As regras utilizadas na implementação foram: Classes_Com_Métodos e Associações_Sem_Nome.

Os erros restantes serão validados manualmente, ou seja, para uma amostragem de modelos conceituais desenvolvidos por aprendizes foram aplicados os algoritmos desenvolvidos como regras para a solução dos erros observados. Os aspectos de validação serão abordado no capítulo VII.

Por fim o capítulo VIII traz as conclusões necessárias e as sugestões para trabalhos futuros.

1.2. Definição do Tema e do Tipo de Pesquisa

A presente pesquisa envolve basicamente conhecimentos da área de engenharia de software (modelagem conceitual da análise de sistemas). Como ferramenta de trabalho foram usados conceitos de linguagens formais e compiladores e sistemas especialistas. A área principal da pesquisa pode ser caracterizada como aplicação da informática no ensino de informática. Esta pesquisa visa investigar ambientes acadêmicos de desenvolvimento e análise de sistemas orientados a objetos.

A proposta está ligada ao desenvolvimento de possibilidades de retorno, por meio informatizado, de informações quando um erro (dos identificados como mais freqüentes na modelagem conceitual) ocorrer.

A pesquisa realizada para a idealização da proposta de dissertação de mestrado baseou-se em modelos conceituais desenvolvidos por aprendizes. Então, muitas vezes, estes trabalhos possuíam alguns erros básicos, ou seja, erros primários, que podem ter sido feitos por inúmeros motivos, entre eles, falta de atenção no momento de construção, má compreensão do assunto abordado e até má compreensão dos conceitos necessários para o desenvolvimento da modelagem.

É importante observar que erros são booleanos, ou seja, verdadeiro ou falso e que, o trabalho trará além de avisos de erro também advertências e sugestões. Também vale salientar que além dos erros propriamente ditos, este trabalho procura identificar situações que seriam mais adequadamente descritas como advertências e também como sugestões.

Os pontos que ficaram obscuros podem ser observados no capítulo que trata das conclusões finais. Estes pontos poderão ser abordados com maior profundidade em trabalhos futuros.

CAPÍTULO II

ORIENTAÇÃO A OBJETOS

2.1. Um Pouco de História

Os conceitos de orientação a objetos surgiram como resultado do desenvolvimento de diversas linguagens e ferramentas de representação que adotaram este paradigma. Não há uma concordância geral sobre tais conceitos, entretanto, cada vez mais vem se firmando um consenso em torno da idéia de um núcleo conceitual canônico extensível (OMG,1994) e (KNUDSEN, 1994).

Nos anos 60 é lançado a Simula-67, que apresentou pela primeira vez os conceitos de classes, rotinas correlatas e subclasses.

Na década de 70, época em que grande parte dos projetos de programação eram não estruturados, ou seja, apresentavam códigos emaranhados, a Xerox-PARC cria a linguagem Smalltalk, considerada a mais pura das LPOO (Linguagem de Programação Orientada a Objetos). No final da década de 80 surge a linguagem C++, uma linguagem híbrida, que reunia conceitos da era das linguagens estruturadas e orientação a objetos. Em meados da década de 90, a orientação a objetos passa a ser incorporada por grande parte dos desenvolvedores de sistemas. A orientação a objeto é uma tecnologia que enxerga os sistemas como sendo uma coleção de objetos interagentes, que envolve todas as atividades de desenvolvimento, entre elas análise, projeto, programação e testes, podendo assim diminuir a diferença semântica entre essas etapas (MUNIZ, 2001).

Atualmente, a orientação a objetos tem se mostrado como uma boa estratégia para o desenvolvimento de sistemas complexos. Entretanto, sua reutilização depende de um conjunto de fatores que precisam ser levados em consideração. A comunidade científica de engenharia de software tem buscado pesquisar estratégias que permitam facilitar o uso da orientação a objetos na produção de software, como acontece em outras engenharias. Neste

contexto, diversas técnicas vêm sendo utilizadas com sucesso no desenvolvimento de produtos de software, embora alguns tópicos ainda necessitem ampla discussão para que possam ter um uso efetivo.

2.2. Vantagens da Orientação a Objetos

São muitas as vantagens da utilização de OO no ciclo de desenvolvimento de um sistema, entre elas podemos destacar: (MARTIN, 1995)

a) Maior facilidade para reutilização de código e por conseqüência de projeto. O reuso pode se dar por herança ou composição. Reuso por herança permite definir uma nova classe, reutilizando outras classes já definidas. Esta reutilização facilita a definição de uma nova classe, bastando definir ou redefinir o que a nova classe acrescenta em termos de funcionalidade. Além disso, cada classe é um pequeno módulo que contém dados e funções. Já o reuso por composição as classes podem ser combinadas para a composição de conjuntos unificados de estruturas e processamento associado (GRAMEIRO, 2002). Com reutilizações se obtém o aumento da qualidade, ganho de confiabilidade, além de que o conhecimento adquirido pode ser compartilhado, reduzindo o esforço de codificação e testes. Desta maneira, este pequeno módulo pode ser reutilizado em outros sistemas.

b) Possibilidades para o desenvolvedor trabalhar em um nível mais elevado de abstração, ou seja, as noções de operação e interface permitem conceber classes de maneira abstrata sem se preocupar em como serão implementadas. Elas diminuem a quantidade de coisas em que temos que pensar (uma operação para vários métodos), o que simplifica a concepção. É graças ao alto nível de abstração que se pode conseguir que o modelo de objetos seja utilizado desde o início da concepção (análise) até a programação. Assim, torna-se mais fácil relacionar qualquer parte do programa com os conceitos mais abstratos que ele implementa. Pode-se relacionar a implementação com as decisões de projetos ("*design decisions*"). Isto é muito

importante para a boa manutenção do programa, tanto durante quanto após o seu desenvolvimento.

c) Graças à interface, o exterior da classe é protegido das modificações que podem acontecer na implementação da classe (interior). Enquanto a interface de uma classe não mudar, a classe não muda para o exterior. Ao mesmo tempo, a implementação pode mudar muito (métodos se tornam atributos, implementação dos métodos podem mudar, etc.).

d) Agrupamento das funções facilita o polimorfismo. Como uma classe contém dados e funções, uma operação pode ter várias implementações em várias classes. Cada implementação é diferente das outras, mas todas pertencem à mesma operação.

e) Com relação à estrutura e funções, o modelo permite dar mais importância à estrutura de um sistema do que às funções. Isso é importante porque as funções que um sistema deve realizar sempre mudam com o tempo, enquanto são muito raras as mudanças na estrutura.

f) Projeto mais rápido: Aplicativos são criados a partir de componentes preexistentes. Muitos componentes são construídos de forma que possam ser customizados para um projeto particular.

g) Integridade: As estruturas de dados podem ser usadas apenas com métodos específicos. Isso é particularmente importante com sistemas cliente/servidor e distribuídos em que usuários desconhecidos poderiam tentar acessar o sistema.

h) Ciclo dinâmico de vida: O alvo do desenvolvimento dos sistemas muitas vezes muda durante a implementação. Ferramentas CASE Orientadas a Objeto tornam as mudanças do ciclo de vida mais fáceis.

2.3. Princípios da Orientação a Objetos

A base da orientação a objetos repousa sobre um pequeno número de conceitos que precisam ser claramente entendidos. Eles são importantes para a compreensão do processo global ao qual se está referenciando.

2.3.1. Objeto

A principal noção de um sistema orientado a objetos e a idéia de objeto. Um objeto é qualquer coisa, real ou abstrata, que é descrita por dados que são de interesse e possuem um comportamento que interessa acompanhar ou é útil. Objetos são entidades autônomas, por princípio. Dois objetos com exatamente os mesmos atributos são dois objetos distintos. Na informática, pode-se pensar na memória de um computador: dois objetos com os mesmos atributos estão em diferentes partes da memória, na verdade eles têm um atributo implícito (o endereço na memória onde eles ficam) que é diferente. Um objeto pode ser uma pessoa, um material, uma instrução, um fato, um lugar, um conceito. Para MUNIZ (2001), ao identificar um objeto é importante considerar:

a) Tudo o que é aprendido pelo conhecimento que não é o sujeito do conhecimento;

b) Tudo o que é manipulável e/ou manufaturável;

c) Tudo o que é perceptível por qualquer sentido;

d) Tudo o que é externo ao sujeito;

e) Um objeto tem um estado, comportamento e identidade;

f) A estrutura e comportamento de objetos similares são definidos em suas classes;

g) Os termos "instâncias" e "objetos" são intercambiáveis.

2.3.1.1. Tipos de Objetos

Os objetos podem ser:

a) Concretos: pessoa, lápis, carro, relógio, etc;

b) Intangíveis: hora, idéias, organização, projeto, etc;

c) Papel: médico, paciente, professor, aluno, etc;

d) Relacional: casamento, parceria, propriedade, etc;

e) Evento: venda, admissão, pane-no-sistema, etc;

f) De Interface: janela, ícone, string de caracteres, etc.

Os objetos possuem propriedades, entre elas pode-se destacar:

a) Estado: é o conjunto das propriedades de um objeto e seus valores;

b) Comportamento: é representado pelas ações e reações do objeto em termos de seu estado e de troca de mensagens;

c) Identidade: é a propriedade que distingue um objeto de outro.

2.3.2. Classes

Classe é a especificação de um conceito no mundo real. As propriedades associadas ao conceito são representadas por atributos (variáveis) e operações (funções).

Há escolas de pensamento que estabelecem uma distinção entre classe e tipo, fazendo correspondência à noção de tipos abstratos de dados (ATKINSON, 1989). Em sistemas baseados em tipos, estes asseguram a correção dos programas, do ponto de vista dos tipos que manipulam, em tempo de execução.

Tipos em linguagens de programação incluem tipicamente arrays, listas, strings e tipos atômicos, enquanto que em linguagem de base de dados os tipos são conjuntos e registros, além dos tipos atômicos.

Uma classe consiste em uma estrutura de dados e em um conjunto de operações aplicáveis à classe, uma vez que a classe pode ser considerada como um modelo para a criação de novos objetos e um meio de permitir que determinadas operações sejam aplicadas sobre os objetos (PALAZZO, 2001).

Uma classe descreve os atributos que seus objetos vão ter e as funções que podem executar um conjunto de objetos que são descritos pelos mesmos dados e possuem o mesmo comportamento. São eles:

a) Conjunto de objetos que compartilham características semelhantes, uma estrutura e um comportamento comum;

b) Os termos "classe" e "tipo" são usualmente (mas não sempre) intercambiáveis. Uma classe pode ser um conceito ligeiramente diferente de um tipo, no qual ela enfatiza a classificação da estrutura e comportamento;

c) Uma classe é um tipo definido pelo usuário que contém o molde, a especificação para objetos, assim como o tipo inteiro contém o molde para as variáveis declaradas como inteiras;

d) A classe associa funções e dados controlando o acesso a estes;

e) Definir classes implica em especificar os seus atributos (dados) e suas funções (código).

2.3.2.1. Subclasse

Subclasse é uma classe que especializa alguns aspectos de outra.

2.3.3. Abstração

Abstração consiste na concentração nos aspectos essenciais em detrimento de detalhes menos importantes para o momento. A abstração denota a característica que a distingue de outros objetos e oferece uma fronteira conceitual claramente definida, sempre a partir da visão do observador. A abstração preserva a liberdade de se tomar decisões evitando, comprometimentos prematuros com detalhes. Ela é o processo pelo qual a mente ou a inteligência compreende o objeto, suas características e sua funcionalidade (MUNIZ, 2001).

2.3.4. Agregação

A agregação é o relacionamento "parte-todo" ou "uma-parte-de" no qual os objetos que representam os componentes de alguma coisa são associados a um objeto que representa a estrutura inteira.

2.3.5. Comportamento

O comportamento é a definição da forma como um objeto age e reage, em termos de mudança de seu estado e envio de mensagem.

2.3.6. Encapsulamento

Também chamado de ocultamento de informações, o encapsulamento é o resultado de ocultar os detalhes de implementação de um objeto. Encapsulamento é o termo formal que descreve a junção de métodos e dados dentro de um objeto de maneira que o acesso aos dados seja permitido somente por meio de alguns de seus métodos. Nenhuma outra parte do

programa pode operar diretamente sobre os dados. A comunicação entre objetos ocorre exclusivamente por meio de mensagens explícitas (MUNIZ, 2001).

2.3.7. Estado

Estado é o resultado cumulativo do comportamento do objeto – uma das condições na qual o objeto existe, caracterizado pelos valores de seus atributos.

2.3.8. Herança

Herança é o fluxo de informações que vai da especificação de uma classe para suas subclasses.

Para PEREZ (2000), herança é sempre transitiva, tal que uma classe pode herdar características de superclasses que estão a muitos níveis acima do seu. Isto é, se a classe *Cachorro* é uma subclasse da classe *Mamífero*, que por sua vez é uma subclasse da classe *Animal*, então *Cachorro* herdará atributos tanto de *Mamífero* quanto de *Animal*. Um fator complicador na noção intuitiva de herança é o fato que subclasse podem sobrescrever comportamentos herdados de superclasses. Por exemplo, a classe *Ornitorrinco* sobrescreve o comportamento de reprodução herdado da classe mamífero, pois ornitorrincos põem ovos.

Utilizando-se do conceito de herança pode-se ter:

a) Códigos com maior capacidade de reusabilidade, fato este que auxilia quando da necessidade de escrever o código;

b) Compartilhamento do código, ou seja, o uso das mesmas classes por desenvolvedores de projetos;

c) Consistência de interface, pois se duas ou mais classes herdam da mesma superclasse, pode-se ter mais certeza de obter uma interface homogênea;

d) Componentes de software podem ser reutilizáveis, ou seja, para uma nova codificação, existirão várias bibliotecas de classes comercialmente disponíveis;

e) Prototipação rápida, ou seja, em um sistema novo, o tempo maior pode ser utilizado em desenvolver os aspectos novos do sistema e não os já existentes;

f) Ocultamento das informações, para o programador que se utiliza de herança, o importante é apenas entender a natureza do componente e sua interface e não os detalhes de como ele foi implementado.

2.3.8.1. Formas de Herança

Para JAIME (2002), as formas mais comuns de utilizar a herança são:

a) Especialização: A subclasse é uma forma especializada da superclasse e satisfaz as especificações da superclasse em todos os aspectos relevantes;

b) Limitação: Ocorre quando o comportamento da subclasse é menor que o da superclasse;

c) Generalização: Nesta situação a subclasse estende o comportamento da superclasse para criar um tipo mais geral de objeto;

d) Especificação: A superclasse pode ser uma combinação de operações não implementadas e que são deixadas para ser implementadas na subclasse, ou seja, a subclasse implementa o comportamento descrito, mas não implementado pela superclasse. Nestes casos a superclasse é também chamada de classe de especificação abstrata;

e) Combinação: É a situação onde uma subclasse pode assumir o comportamento de duas ou mais superclasses. É também conhecida como herança múltipla. Um exemplo aqui seria o de um monitor de estudos, que hora comporta-se como aluno e hora comporta-se como mestre.

2.3.9. Mensagem

Os objetos se comunicam entre si através de mensagens que emitem e recebem. Isto permite definir uma interface precisa entre o objeto e seu ambiente.

Diferente dos dados passivos em sistemas tradicionais, os objetos possuem habilidade para agir. A ação ocorre quando um objeto recebe uma mensagem, isto é, uma solicitação para que se comporte de uma determinada maneira. Quando os programas orientados a objeto são executados, os objetos estão recebendo, interpretando e respondendo mensagens de objetos (WINBLAD, 1993).

2.3.10. Módulo

É uma unidade de programa que contém declarações, expressas num vocabulário de uma linguagem de programação particular, que forma a realização física de uma parte ou de todas as classes e objetos do projeto lógico do sistema. É uma unidade de códigos que serve como um bloco construído para uma estrutura física de um sistema. É composto das seguintes partes:

Interface: É a visão externa de uma classe, objeto ou módulo, os quais enfatizam suas abstrações enquanto esconde suas estruturas e os segredos de seus comportamentos.

Implementação: É a visão interna de uma classe, objeto ou módulo, incluindo os segredos de seu comportamento.

2.3.11. Polimorfismo

O termo "polimórfico" tem origem grega e significa "muitas formas". Em orientação a objetos é o nome dado a capacidade que diferentes objetos têm de responder a uma mesma mensagem. Sobrecarga e coerção são formas de polimorfismo ad-hoc e polimorfismo puro e polimorfismo parametrizado ou genérico são formas de polimorfismo universal.

Em se falando de polimorfismo ad-hoc, não existe um modo único e sistemático de determinar o tipo de resultado de uma função em termos dos tipos dos seus argumentos de entrada. Esta é uma forma limitada de polimorfismo, pois trabalha com um número limitado de tipos.

Já o polimorfismo universal existe uma forma sistemática e única de trabalhar com um número potencialmente infinito de tipos (MARCELO, 2002). O polimorfismo é possível graças à existência de variáveis polimórficas. Uma variável polimórfica é aquela que pode armazenar valores de tipos diferentes. Estas variáveis englobam o princípio da substituição. Em outras palavras, enquanto há um tipo esperado para qualquer variável, o tipo real pode ser aquele de qualquer valor que seja subtipo do tipo esperado.

Em linguagens tipadas dinamicamente, tais como Smalltalk e Objective-C, todas as variáveis são potencialmente polimórficas, qualquer variável pode armazenar valores de qualquer tipo, neste caso o tipo esperado é definido pelo conjunto de comportamentos esperados.

Já nas linguagens tipadas estaticamente, tais como C++ e Java, a situação é ligeiramente mais complexa, ele ocorre nestas linguagens através da diferença entre o tipo estático de uma variável (aquela da declaração) e o tipo do valor (dinâmico) que a variável contém.

2.3.11.1. Sobrecarga

O polimorfismo de sobrecarga ocorre quando um nome de função está associado a mais de dois corpos da função, ou seja, quando um nome de função for usado mais de uma vez com diferentes tipos de parâmetro. Por exemplo, uma função soma pode ser sobrecarregada pra operar com dois parâmetros: inteiros e reais. Isto é, definimos duas funções soma com tipos de parâmetros de entrada diferentes. A informação sobre o tipo de parâmetro é utilizada para selecionar a função apropriada.

2.3.11.2. Sobrecarga e Sobrescrição

São os mecanismos de substituição e refinamento, já definido anteriormente, porém é importante observar alguns pontos. Em uma superclasse abstrata, há um método geral definido para uma passagem particular que é herdado e usado pela subclasse. Em, pelo menos, uma subclasse, um método com o mesmo nome é definido, o qual esconde o acesso

ao método geral para instâncias desta classe. Dizemos então que o segundo método sobrescreve o primeiro.

Como o primeiro e o segundo método possuem o mesmo nome, sempre que há sobrescrição há sobrecarga. Entretanto, o inverso não é verdadeiro. Sobrecarga pode ocorrer pela presença de dois ou mais métodos compartilhando o mesmo nome e uma mesma classe que não herda nada de outra, neste caso não há sobrescrição. Também, métodos sobrescritos ou sobrecarregados não necessitam obrigatoriamente possuir semelhança de comportamento, isto é, dois métodos que compartilham o mesmo nome ou onde um esconde outro por sobrescrição, podem realizar ações completamente diferentes.

2.3.11.3. Coerção

Algumas linguagens de programação permitem que haja uma conversão implícita entre certos tipos de dados. Estas conversões implícitas são realizadas automaticamente pelo compilador e são denominadas coerção. Em geral, coerções são limitadas a situações onde não há perda de informação, como na conversão de um valor inteiro para um valor real correspondente.

Coerção proporciona um meio de contornar a rigidez de tipos monomórficos. Se um contexto particular demanda um determinado tipo e um tipo diferente é passado, então a linguagem verifica se existe uma coerção adequada.

2.3.11.4. Polimorfismo Puro

Muitos autores reservam o termo polimorfismo puro (ou de inclusão) para situações onde uma função pode ser usada com uma variedade de argumentos. Note que, neste caso, os parâmetros da função são polimórficos e não o nome da função, como é o caso com sobrecarga.

A capacidade para criar funções polimórficas é uma das técnicas mais potentes da programação orientada a objetos, pois esta técnica permite que o código escrito apenas uma

vez, em nível mais alto de abstração, seja ajustado para se adequar a uma variedade de situações. Usualmente, o programador efetua esse ajuste através do envio de outras mensagens para o receptor. Estas mensagens subseqüentes não estão freqüentemente relacionadas com a classe no nível do método polimórfico, ao invés disso, são métodos virtuais definidos nas classes de nível inferior.

2.3.11.5. Genericidade

Uma outra forma de polimorfismo é fornecida pela facilidade conhecida como genericidade. Genéricos oferecem uma maneira de parametrizar uma classe ou uma função através do uso de um tipo, assim como parâmetros normais para uma função oferecem uma forma de definir um algoritmo abstrato sem identificar valores específicos.

2.3.12. Instância

Instância se refere implicitamente a uma classe, é sempre uma instância de uma classe específica. Objetos podem ser considerados independentes de suas classes. Qualquer conceito abstrato pode ter instâncias (aplicações concretas do conceito). Por exemplo, um método pode ser considerado uma instância de uma operação; um atributo particular de uma classe pode ser considerado uma instância do conceito geral de atributos, etc.

CAPÍTULO III

UML (Unified Modeling Language)

3.1. Introdução

A Unified Modeling Language (Linguagem de Modelagem Unificada) – UML é uma notação (principalmente diagramática) para modelagem de sistemas, usando conceitos de orientação a objetos (LARMAN, 2000). A UML também pode ser definida como uma linguagem para especificação, visualização, construção e documentação de artefatos de sistemas de software. Ela representa uma coleção das melhores experiências na área de modelagem de sistemas orientados a objetos, as quais tem obtido sucesso na modelagem de grandes e complexos sistemas (RATIONAL, 2000).

A notação UML é muito rica, tanto na "largura" (se pode modelar uma grande variedade de sistemas como sistemas de tratamento de dados, sistemas de tempo real e sistemas distribuídos), como em "profundidade" (pode-se modelar muitos detalhes, cada noção tem várias nuanças).

A notação UML surgiu de um esforço coletivo para a junção dos métodos Booch de Grady Booch, OOSE, também conhecida por Objectory de Ivar Jacobson e OMT de James Rumbaugh, além das idéias de outros metodologistas, como é descrito na figura 01.

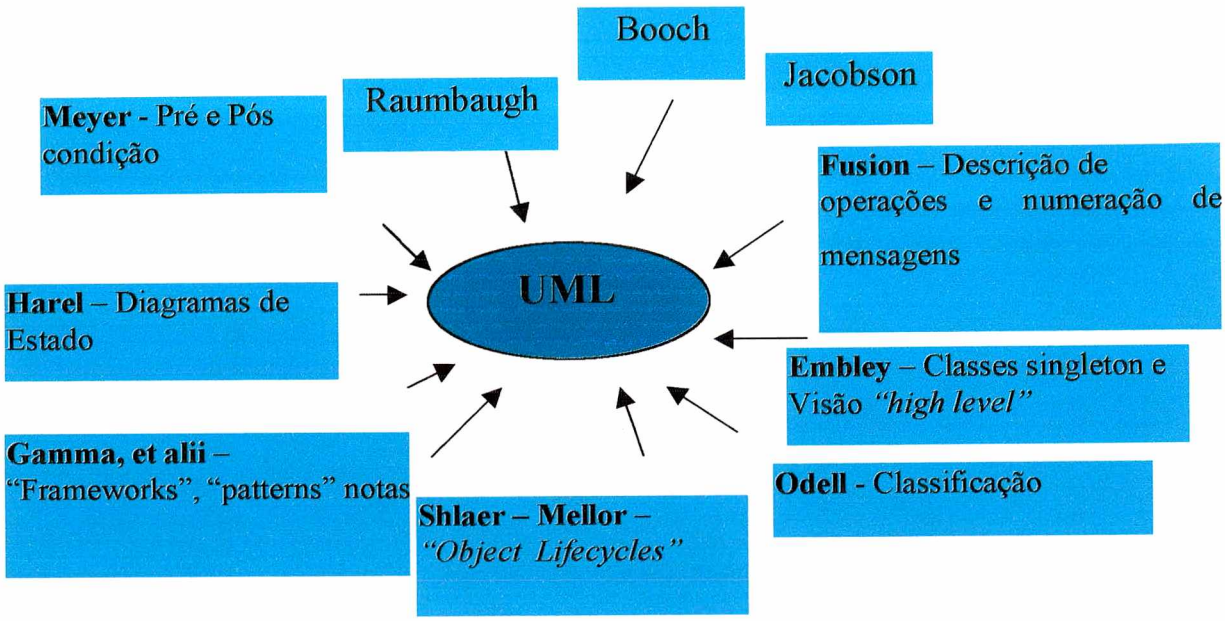


Figura 01- Convergência de tecnologias para UML

A notação UML pode se usada para:

a) Mostrar os limites de um sistema e suas principais funções, usando casos de uso e agentes;

b) Ilustrar a realização de casos de uso, usando os diagramas de interação;

c) Representar a estrutura estática de um sistema, usando diagramas de classe;

d) Modelar o comportamento de objetos, com diagramas de estado e colaboração;

e) Apresentar a implementação física e a arquitetura de um sistema, com diagramas de componentes;

f) Criar novas formas de expressões usando estereótipos.

Conforme RATIONAL (2000), Booch, Jacobson e Rumbaugh adotaram quatro metas para este processo de unificação:

a) Representar sistemas completos (em vez de somente porções de software) usando conceitos de orientação a objetos;

b) Estabelecer uma ligação explícita entre conceitos e código executável;

c) Levar em conta os fatores de escala que são inerentes aos sistemas complexos e críticos;

d) Criação de uma linguagem de modelagem utilizável por homens e máquinas.

3.2. Definindo termos na notação UML

Na notação UML, são usados os termos "classe" e "tipo", mas, não "conceito". Não há um consenso universal sobre o significado de classe e tipo. Assim, para evitar ambigüidades, a notação UML define precisamente esses termos conforme usados no seu metamodelo (o modelo da própria UML), embora sejam usadas definições alternativas contraditórias por outros autores e praticantes. Alguns praticantes, por exemplo, preferem usar a palavra conceito quando estão se referindo ao modelo conceitual e a palavra classe para quando estiverem se referindo ao diagrama de projeto, por achar que o conceito de "tipo" fica restrito à linguagem de programação, pois o tipo refere-se às variáveis e não aos objetos denotados por elas. Neste contexto, o tipo da variável pode ser "Número" mas a classe do objeto denotado pela variável poderia ser "NúmeroInteiroNegativo".

Independente da definição, o aspecto importante a ser notado é que é útil fazer a distinção entre a perspectiva de um analista de domínio, procurando conceitos no mundo real, tal como uma venda, e a perspectiva de projetistas de software, especificamente entidades de software, tais como uma classe Venda.

A notação UML pode ser usada para ilustrar ambas as perspectivas com notação e terminologia bastante similares, de maneira que é importante ter em mente qual a perspectiva que está sendo assumida (ou uma visão de análise, de projeto ou de implementação).

Para (FURLAN, 1998), uma **classe** é um modelo utilizado para criar objetos específicos e relacionados, uma descrição de um conjunto de instâncias que compartilham as mesmas operações, atributos, relacionamentos e semântica. Uma **instância** de uma classe é um objeto real que possui dados e aguarda por executar serviços. A idéia de conceito também está presente falando-se de classe. Para (LARMAN, 2000), o conceito é uma classe que não possui métodos. As figuras 02 e 03 demonstram os comentados acima. Um método é a implementação de uma operação que especifica o algoritmo ou o procedimento usado pela operação.

Alguns autores restringem a definição de uma classe, como sendo uma implementação concreta de software, ou como artefatos de projeto, enquanto os conceitos são artefatos de análise tal como uma classe (FOWLER, 1996). Contudo, na notação UML, o termo é tratado mais amplamente, abrangendo especificações que precedem a implementação. Na notação UML, uma classe de software implementada, é chamada, de modo mais específico, uma classe de implementação.

Na notação UML, uma operação é "um serviço solicitado a um objeto para produzir um comportamento" (BJR, 1997), e um método é "a implementação de uma operação que especifica o algoritmo ou o procedimento usado pela operação".

A definição da notação UML para tipo é similar àquela para uma classe – ela descreve um conjunto de objetos semelhantes com atributos e operações – mas ela não pode incluir métodos. A implicação disso é que um tipo é uma especificação de uma entidade de software, e não de uma implementação. Isso também implica que um tipo, em UML, é independente da linguagem de programação.

O termo interface é definido como um conjunto de operações externamente visíveis. Em termos de notação UML, eles podem, estar associados a tipos e classes (e a pacotes – os quais agrupam elementos). Embora conceitos do mundo real possam ter uma interface (por exemplo, a interface de um telefone), o termo é tipicamente usado no contexto de uma interface para entidades de software (LARMAN, 2000).

Figura 02 – Conceito

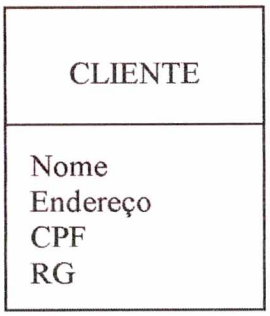


Figura 03 – Classe

3.3. Elementos Essenciais da notação UML

Para que a modelagem baseada na notação UML seja eficiente, ela deve possuir alguns elementos essenciais, entre eles, destacam-se os elementos estruturais, elementos comportamentais, elementos de agrupamento e elementos de extensão e anotação.

Nos elementos estruturais, dá-se importância às classes, aos objetos, aos relacionamentos e aos componentes. Já nos elementos comportamentais, a importância está relacionada aos casos de uso (use case), aos estados, às mensagens e transições e aos agentes. Para os elementos de agrupamento o que importa são os pacotes e aos elementos de extensão e anotação, os estereótipos, as restrições, as *tagged values* e as anotações.

3.4. As Fases de Desenvolvimento da UML

Muitos autores (LARMAN, 2000), (LARMAN, 2002), (FURLAN, 1998) e (FOWLER, 1996), entre outros, definem as fases de desenvolvimento usadas na notação UML conforme o seu entendimento do modelo, porém, este estudo será fundamentado na metodologia definida por LARMAN (2000). Por ser uma das mais recentes é muito utilizada nos cursos de Ciência da Computação, e, por tanto, mais adequada para o caso.

O processo de desenvolvimento pode ser dividido em fases que demonstram em cada etapa do modelo sua importância, podendo, assim, dividir o trabalho em partes e fazer com que ele possa ser mais bem compreendido. LARMAN (2000) destaca em seu modelo as fases de planejar e elaborar, analisar, projetar e construir.

Na fase planejar e elaborar, primeira fase identificada, tem-se a presença da descrição do problema, dos casos de uso e do diagrama de casos de uso (*use case*).

Um **caso de uso** é um documento narrativo que descreve de forma estruturada a seqüência de eventos de um ator (um agente externo) que usa um sistema para completar um processo (JACOBSON, 1992).

LARMAN (2000) aponta para o fato de que casos de uso são importantes para compreender os processos do domínio e o ambiente externo (fatores externos que participam dos processos). É sempre bom começar um projeto com um caso de uso de alto nível, ou seja, um relato dos principais passos, que descreve o evento de forma breve e sucinta. Com o passar das etapas do projeto, os casos de uso de alto nível são desmembrados e esmiuçados, sendo descritos de forma detalhada, passo a passo. O uso de casos de uso tem vários objetivos importantes, entre eles podemos destacar:

a) Ser compreensíveis para usuários que provavelmente não entendem de informática;

b) Incentivar a análise do sistema, especificando as funcionalidades necessárias;

c) Delimitar o sistema;

d) Servir de base para os casos de testes;

Falando de casos de uso, é importante também definir o sistema e suas fronteiras, ou

seja, deixar bem claro o que é externo e o que é interno ao sistema, e quais as suas responsabilidades, por exemplo, se uma loja for a fronteira, somente o cliente é o ator, mas se apenas o hardware e o software forem fronteira, o caixa também passa a ser ator.

O exemplo de um caso de uso de alto nível, mostrado abaixo pode melhor especificar o anteriormente citado.

Caso de Uso: **Comprar itens em uma loja de conveniência.**

Atores: Cliente e Caixa

Tipo: Primário

Descrição: Um cliente chega a um ponto de pagamento, com os itens que deseja comprar. O caixa registra os itens de compra e recebe o pagamento. Ao término, o cliente sai da loja com os itens.

Já os **diagramas de casos** de uso, ilustram um conjunto de casos de uso para um sistema, os atores e sua relação com os casos de uso. Esses diagramas mostram como o sistema a ser desenvolvido vai interagir com o seu ambiente (usuários e/ou outros sistemas). Ele é importante por ser a base do processo de desenvolvimento do sistema, especificando a estrutura do domínio e do sistema. Esse diagrama é diferente dos outros, pois contém poucos gráficos. A figura 04 demonstra este conceito levando em consideração o caso de uso descrito acima.

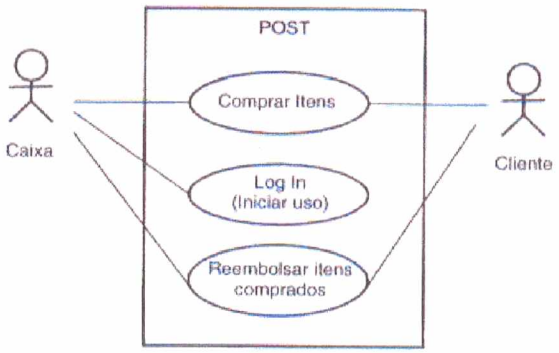


Figura 04 – Diagramas de Caso de Uso (LARMAN, 2000)

Na segunda fase, ou seja, na fase analisar, temos a construção do **modelo conceitual**, que é um conjunto de diagramas de estrutura estática, nos quais não se definem operações, e por esse ser parte do enfoque principal do trabalho será abordado com maior precisão no próximo capítulo. Também surge a necessidade de registrar termos em um glossário, que também pode ser chamado de dicionário do problema. É nele que se lista e define todos os termos que requerem esclarecimentos, de modo a melhorar a comunicação e a reduzir o risco de mal entendidos. Manter o glossário é uma atividade contínua durante todo o projeto. Uma proposta de glossário é definida por (LARMAN, 2000) no quadro 01.

Termo	*Categoria*	*Comentário*
Comprar Itens	Caso de Uso	Descrição do processo de um cliente comprando itens em uma loja
EspecificaçãodeProduto.descrição :Text	Atributo	Uma curta descrição de um item em uma venda e sua EspecificaçãodeProduto associada
Item	Tipo	Um item de venda em uma loja
Pagamento	Tipo	Um pagamento em dinheiro
EspecificaçãodeProduto.preço :Quantidade	Atributo	O preço de um item em uma venda e sua EspecificaçãodeProduto associada
LinhadeItemdeVenda.quantidade :Integer	Atributo	A quantidade comprada de um tipo de item
Venda	Tipo	Uma transação de vendas
LinhadeItemdeVenda	Tipo	Uma linha de item para um particular item comprado numa venda

Termo	*Categoria*	*Comentário*
Loja	Tipo	O local onde ocorre venda de itens
Venda.total :Quantidade	Atributo	O total geral da venda
Pagamento.quantia :Quantidade	Atributo	A quantidade de dinheiro fornecida ou apresentada para pagamento pelo cliente
EspecificaçãodeProduto.upc :UPC	Atributo	O código universal de produto do Item e sua EspecificaçãodeProduto

Quadro 01 – Glossário

Outro ponto importante é o **diagrama de seqüência do sistema**, que ilustra os eventos gerados pelos atores que são reconhecidos pelo sistema. Ele mostra, uma seqüência particular de eventos dentro de um caso de uso. Um diagrama de seqüência oferece uma visão detalhada de um caso de uso. Ele mostra uma interação organizada em forma de seqüência dentro de um determinado período de tempo e contribui para que se processe a documentação do fluxo de lógica dentro da aplicação. Esse momento é considerado parte da investigação sobre qual sistema deve ser construído. Cabe aqui ressaltar a diferença entre um evento de sistema e uma operação de sistema. O primeiro diz respeito a um evento externo de entrada gerado por um ator para um sistema. O evento inicia uma operação de resposta do sistema.

Já uma operação do sistema é uma operação executada em resposta a um evento do sistema. Pode-se observar melhor esse exemplo na figura 05 definida por (LARMAN, 2000).

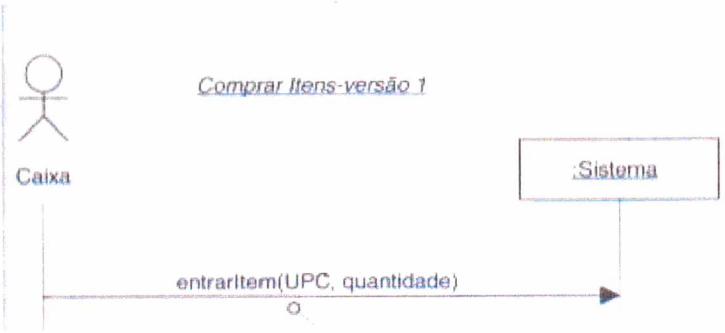


Figura 05 – Evento do Sistema

Nesta mesma fase temos também os **contratos** que auxiliam a definir o comportamento do sistema e definem o efeito das operações sobre o sistema. No geral, um contrato é um documento que descreve o que uma operação se compromete a atingir. Normalmente, segue um estilo declarativo, enfatizando o que acontecerá, em vez de como será conseguido. É comum serem expressos em forma de estados definidos por pré-condições e pós-condições. Um exemplo de contrato segue no quadro 02.

Nome	Nome da operação e parâmetros
Responsabilidades	Uma descrição informal das responsabilidades que esta operação deve cumprir.
Tipo	Nome do tipo (conceito, classe de software, interface).
Referências Cruzadas	Números de referência de funções do sistema, casos de uso.
Notas	Notas de projeto, algoritmos e assim por diante.
Exceções	Casos excepcionais.
Saída	Saídas que não são para a Interface do usuário, tais como mensagens ou registros enviados para fora do

	sistema.
Pré-Condições	Hipóteses e assertivas sobre o estado do sistema antes da execução da operação.
Pós-Condições	O estado do sistema após a operação ter sido completada

Quadro 02 - Contrato

Na terceira fase, ou seja, na fase projetar, destaca-se a inserção dos diagramas de colaboração. Um **diagrama de colaboração**, que é um diagrama de interação assim como o diagrama de seqüência, e que ilustra as interações entre objetos em forma de grafo ou rede, nele pode-se expressar mais informações contextuais, tais como o tipo de visibilidade entre objetos. Além disso, facilita a expressão de condições lógicas e execução concorrente ("paralela" ou "assíncrona"). Seu foco principal não é o tempo, mas a organização dos objetos, por isso mostra explicitamente as conexões entre os objetos. A figura 06 , descrita por FURLAN (1998) demonstra melhor esta relação.

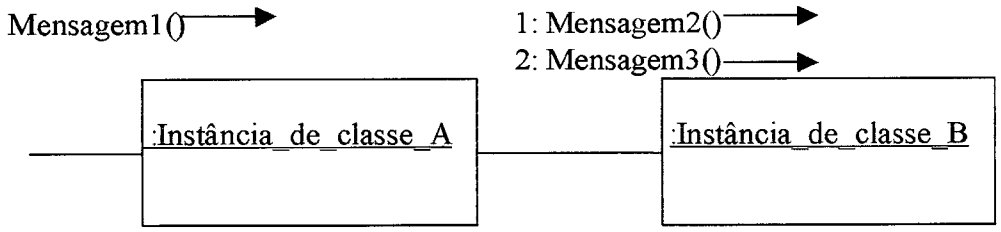


Figura 06 – Diagrama de Colaboração

É importante também nesta fase determinar a **visibilidade**, que é a habilidade de um objeto "ver", ou de ter e reter uma referência para outro objeto. Em termos mais gerais, ela está relacionada com o problema do escopo. Existem quatro maneiras comuns pelas quais a visibilidade de um objeto A para um objeto B pode ser conseguida:

a) Visibilidade por Atributo: B é um atributo de A;

b) Visibilidade por Parâmetro: B é um parâmetro de um método de A;
c) Visibilidade Localmente Declarada: B é declarado com um objeto local em um método de A;
d) Visibilidade Global: B é, de alguma forma, globalmente visível.

Neste ponto surgem os padrões para atribuições de **responsabilidades**. Booch e Rumbaugh, (BJR, 1997) definem responsabilidade como um contrato ou obrigação de um tipo ou classe. As responsabilidades estão relacionadas às obrigações de um objeto em termos de seu comportamento. Basicamente, essas responsabilidades são "conhecer" e "fazer". As responsabilidades de "fazer" de um objeto incluem fazer algo a ele próprio, iniciar ações em outros objetos, controlar e coordenar atividades em outros objetos. As responsabilidades de "conhecer" de um objeto incluem conhecer dados privados encapsulados, conhecer objetos relacionados, conhecer coisas que ele pode derivar ou calcular.

Um padrão é uma descrição conhecida de um problema e a sua solução, que pode ser utilizado em novos contextos. Em termos ideais, ele fornece aconselhamento sobre como utilizá-lo em circunstâncias variáveis (LARMAN, 2000). Mais simplesmente, um padrão é um já nomeado problema/solução, que pode ser utilizado e com conselhos de como utiliza-lo em novas situações.

Larmam apresenta como padrões de responsabilidade os padrões GRASP (*General Responsability Assignment Software Patterns*), que define entre os mais importantes, o *Expert* (especialista), *Creator* (criador), *Low Couping* (acoplamento fraco), *High Cohesion* (alta coesão) e *Controller* (controlador).

O padrão *Expert* (especialista) é o padrão que atribui responsabilidade ao especialista da informação. Ele conduz a projetos onde um projeto de software faz aquelas operações que normalmente são feitas pela coisa do mundo real que ele representa. No mundo real, por exemplo, a palavra venda, não informa o valor total dessa venda, já em software a venda pode assumir a responsabilidade de informar o valor.

O padrão de responsabilidade *Creator* (criador), é responsável por criar instâncias de classes. A criação de objetos é uma das atividades mais comuns em um sistema orientado a objetos. Neste sentido, é útil ter um princípio geral para a atribuição destas

responsabilidades de criação. Sendo estas responsabilidades bem atribuídas, o projeto apresentará acoplamento fraco, melhoria na clareza, encapsulamento e reutilização.

Já o padrão de responsabilidades *Low Couping* (acoplamento fraco) é a maneira de fazer com que exista baixa dependência e aumentar a reutilização. Para o padrão *High Cohesion* (alta coesão), Booch descreve como algo existente para quando os elementos de um componente (tal como uma classe) "trabalham todos em conjunto" para fornecer algum comportamento bem delimitado (BOOCH, 1994), aumentando a clareza e a compreensão do projeto.

O *Controller* (controlador) é quem atribui a responsabilidade do tratamento de uma mensagem de um evento.

A última fase é a fase **construir,** que é a hora de mapear projetos para códigos, em uma linguagem orientada a objetos ou não orientada a objetos. Durante as fases planejar/elaborar, analisar e projetar, muitos pontos que eram obscuros aos elementos que compunham a equipe de desenvolvimento podem ser desvendados, facilitando a geração do código fonte, porém é bem possível que ainda que as fases anteriores à codificação fossem extremamente bem elaboradas, problemas de codificação possam existir, então essa fase não será um trabalho simples, mas outro momento de concepção, de criação do produto final.

CAPÍTULO IV

O MODELO CONCEITUAL

4.1. Características Gerais

Os modelos conceituais podem representar inúmeros sistemas da natureza. No sistema solar temos os conceitos de planetas, órbita, força gravitacional entre outros. A Física utiliza-se das notações matemáticas para representar os conceitos que descrevem o sistema solar. A Língua Portuguesa também pode ser utilizada para descrever conceitos de um domínio. Na Biologia, os conceitos referentes às espécies, gênero família, classes, ordem, filo e reino dos seres vivos são descritos em Português e Latim (LEITE, 2001).

A especificação que melhor retrata os conceitos de orientação a objetos na análise ou na investigação é a decomposição do problema em conceitos e objetos individuais – as coisas nas quais está-se interessado.

Para os diversos escritores da área existem poucas divergências sobre o significado de modelo conceitual. Segundo (FOWLER, 1996), um modelo conceitual é uma representação de conceitos do domínio do problema. Já para (LARMAN, 2000), um modelo conceitual é uma ferramenta de comunicação.

Em UML, um modelo conceitual é exibido como um conjunto de diagramas de estrutura estática, nos quais não se definem operações. O termo modelo conceitual tem a vantagem de enfatizar fortemente os conceitos do domínio e não de entidades de software. Baseado nos conceitos citados acima pode-se afirmar que o modelo conceitual na UML é o mesmo que a "representação de coisas do mundo real" do domínio do problema, não de componentes de software, como uma classe de software.

A modelagem conceitual é considerada a fase crucial do processo de projeto. Durante essa fase, o projetista deve compreender, estruturar e representar, através de um modelo de

dados de alto nível, a semântica do mundo real de modo que esse modelo reflita as necessidades de usuários de forma natural e direta. Construir um bom esquema conceitual requer bastante cuidado e um entendimento detalhado do domínio da aplicação a ser modelada (MARINHO, 2001).

4.2. Conceitos

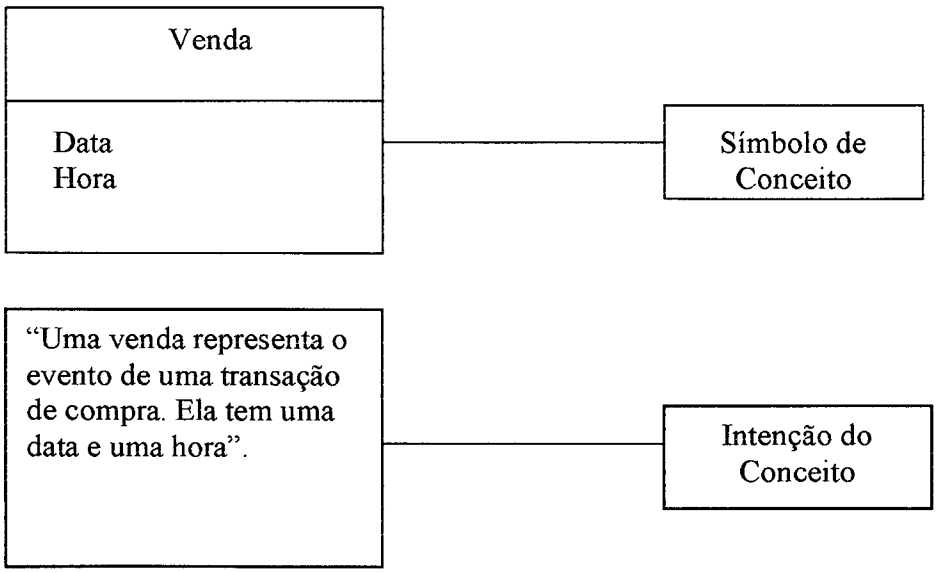
De modo informal, um conceito é uma idéia, uma coisa ou um objeto. Mais formalmente, um conceito pode ser considerado em termos do seu símbolo, da sua intenção e da sua extensão.

a) Símbolo: palavras ou imagens representando um conceito;

b) Intenção: a definição de um conceito;

c) Extensão: o conjunto de exemplos aos quais o conceito se aplica.

Podemos observar estes conceitos através da figura 07 que foi representada por LARMAN (2000).



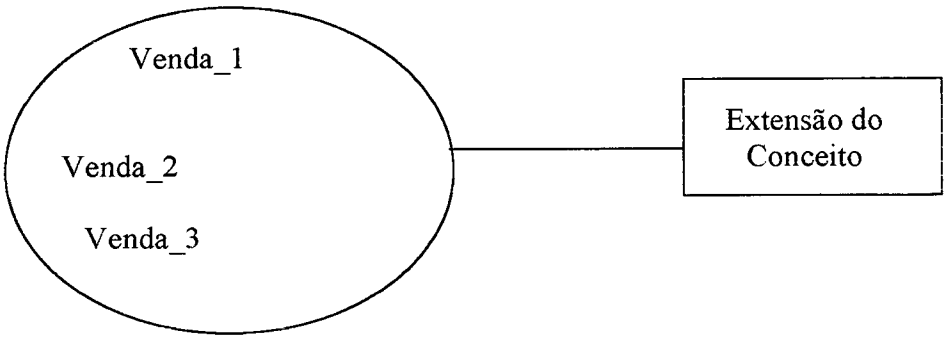


Figura 07 – Um conceito com símbolo, intenção e extensão (LARMAN, 2000)

O conceito, dentro da análise orientada a objetos é a dimensão para decomposição de problemas de softwares complexos. Depois desses conceitos do domínio identificados, documenta-se os resultados em um modelo conceitual.

Porém, para que os conceitos sejam identificados corretamente torna-se necessário observar alguns pontos como:

a) É melhor especificar em excesso um modelo conceitual com muitos conceitos de granularidade fina do que subespecificá-lo;

b) É importante não excluir conceitos, simplesmente pelo fato de que os requisitos não indiquem sua necessidade, ou pelo fato de que ele não contenha atributos;

c) Identificar os substantivos e as frases que podem estar no lugar de um substantivo nas decisões textuais do domínio de um problema e considerá-las, (as frases), como candidatas para serem incluídas no modelo conceitual.

Neste último, ao utilizar o conceito definido, deve-se tomar cuidado, pois não é possível um mapeamento mecânico de substantivos para conceitos e as palavras numa linguagem natural tendem a ser ambíguas (principalmente na língua inglesa). (LARMAN, 2002).

4.3. Nomes no Modelo Conceitual da notação UML

No modelo conceitual da notação UML costuma-se usar a mesma técnica que os cartógrafos utilizam na produção de mapas. Eles desenvolvem os seus trabalhos baseados em três passos principais:

a) Sempre usando nomes existentes no território. Em modelagem conceitual isso significa usar nomes do domínio para conceitos e atributos;

b) Excluindo características irrelevantes. Desta forma, no modelo conceitual pode-se excluir conceitos do domínio do problema não pertinentes aos requisitos;

c) Não incluindo coisas que não estão no território, ou seja, para a modelagem conceitual não tem lógica colocar coisas que não estão no domínio do problema.

4.4. Adicionando Associações no Modelo Conceitual da notação UML

Uma associação é uma operação que descreve um conjunto de vínculos entre elementos do modelo. As classes se associam entre si. Essas associações se apresentam de diferentes formas, entre elas a associação unária e binária, que são de longe as mais importantes, e a ternária.

Uma associação é usada para representar uma dependência estrutural entre objetos, geralmente de classes diferentes. Ela pode possuir um nome para clarificar a natureza da relação entre a associação, bem como cada terminação da associação pode ditar papéis para mostrar como uma classe é vista pela outra.

Quando estamos construindo um modelo conceitual em UML, é importante que as associações de conceitos satisfaçam os requisitos de informação que os casos de uso determinaram para melhor compreensão do modelo conceitual.

Em UML, as associações são descritas como "relacionamentos estruturais entre objetos e tipos distintos". A figura 08 demonstra com melhor clareza este conceito.

Registra-a-Corrente

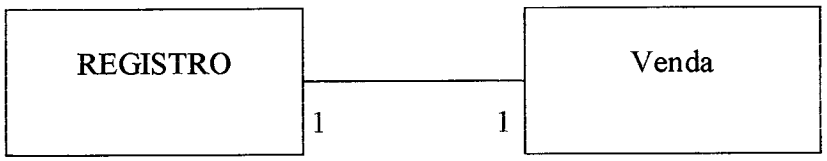


Figura 08 - Associações

Uma associação é representada como uma linha entre conceitos, com um nome. A associação é inerentemente bidirecional, significando que é possível o percurso lógico dos objetos de um dos tipos para o(s) do(s) outro(s) e vice-versa. As pontas de uma associação podem conter uma expressão de multiplicidade, indicando o relacionamento numérico entre as instâncias dos conceitos.

Covencionou-se fazer uma leitura da associação da esquerda para a direita ou de cima para baixo, embora na UML isso não seja uma regra, pois na UML pode-se utilizar um triângulo (▷)[1] para indicar a leitura.

No quadro 03, citado em (LARMAN, 2000), podemos encontrar uma lista de associações comuns, porém que valem a pena ser consideradas. Para o desenvolvimento desta lista de associações, utilizou-se como exemplo o domínio de uma loja de reservas de passagens aéreas.

CATEGORIA	EXEMPLOS
A é uma parte física de B	Asa - Aeronave
A é uma parte lógica de B	PernadeVôo – RotadeVôo
A está fisicamente contida em /sobre B	Passageiro – Aeronave
A está logicamente contida em B	Vôo – ProgramaçãodeVôo
A é uma descrição de B	DescriçãodeVôo – Vôo
A é uma linha de item de uma	ServiçodeManutenção

[1] Neste caso, da esquerda para a direita.

transação ou relatório B	RegistrodeManutenção
A é conhecido / "logado"/ registrado/ relatado/ capturado por B	Reserva – ManifestodeVôo
A é um membro de B	Piloto – LinhaAérea
A é uma subunidade organizacional de B	Manutenção – LinhaAérea
A usa ou gerencia B	Piloto – Aeronave
A se comunica com B	AgentesdeReseva – Passageiros
A está relacionado com uma transação B	Passageiro – Bilhete
A é uma transação relacionada com uma outra transação de B	Reserva – Cancelamento
A está em seguida / adjacente/ visinho a B	Cidade – Cidade
A é possuído por B	Avião - LinhaAérea

Quadro 03 – Lista de Associações Comuns

Ainda pode-se dizer que cada extremo de uma associação é chamado de papel. Os papéis de uma associação possuem a função de apresentar informações sobre a associação, entre elas multiplicidade, nome das associações, ordenação, qualificador, agregação/composição, navegabilidade, especificador de interface, mutabilidade, visibilidade e generalização. Destes aspectos é importante salientar: multiplicidade, nome da associação, ordenação, qualificador, agregação/composição, navegabilidade, especificador de navegabilidade, mutabilidade, visibilidade e generalização.

4.4.1. Multiplicidade

A multiplicidade pode ser opcional ou, obrigatória e singular ou múltipla. Os quatro tipos resultantes são: opcional singular: 0..1, obrigatória singular: 1, opcional múltipla: 0..* e obrigatória múltipla: 1..*.

Como exemplo de multiplicidade em associações do modelo conceitual, pode-se observar o quadro que se encontra em (FURLAN, 1998).

MULTIPLICIDADE	SIGNIFICADO
0..1	Zero ou Um
1	Somente Um
0..*	Maior ou igual a Zero
* 1..*	Maior ou igual a Zero Maior ou igual a Um
1..15	De Um a Quinze (inclusive)
1..2; 7..15; 19; 23..*	De Um a Dois, de Sete a Quinze, Dezenove ou acima de Vinte e Três (inclusive)

Quadro 04 – Multiplicidade no Modelo Conceitual da UML

4.4.2. Nome da Associação

Nas associações costuma-se colocar nomes próximos a linha que representa o relacionamento. Freqüentemente o nome atribuído é um verbo, muito embora também possam ser utilizados substantivos. Cabe sempre lembrar que esses nomes devem estar relacionados ao domínio do problema ao qual se está modelando. A figura 09 descreve o conceito (FURLAN, 1998).

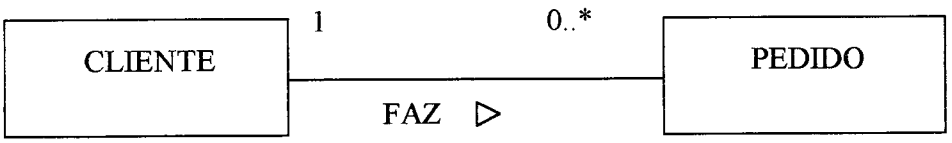


Figura 09 - Exemplo de Associação entre Conceitos (FURLAN, 1998)

4.4.3. Ordenação

Se a multiplicidade é maior que um, o conjunto de elementos relacionados é ordenado ou não ordenado. Os elementos são ordenados em uma lista, cuja especificação genérica inclui todos os tipos de ordenação podendo ser especificados por uma restrição de palavra-chave {ordenado}. Um relacionamento ordenado pode ser implementado de vários modos, mas isso normalmente é especificado como uma propriedade de geração de código especificada na linguagem de programação para selecionar uma implementação particular. A figura 10 demonstra um exemplo de ordenação.

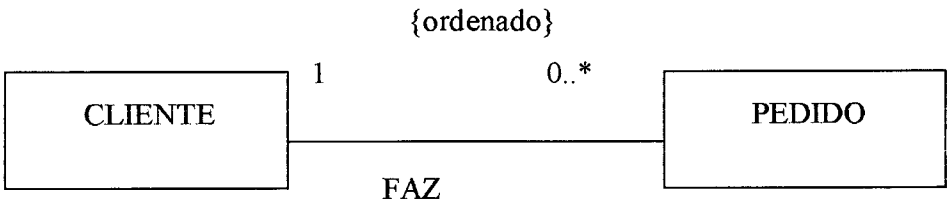


Figura 10 - Exemplo de Ordenação de Associações na UML

A declaração {ordenado} não especifica como a ordenação é estabelecida ou mantida, operações que inserem elementos novos tem que fazer provisão para especificar sua posição implícita ou explicitamente.

4.4.4. Qualificador

Uma associação qualificada é o equivalente na UML ao conceito de programação conhecida como matrizes, mapas e dicionários associativos. Um qualificador é um atributo de associação ou estrutura de atributos cujos valores particionam um conjunto de objetos relacionados a um objeto por associação. É empregado para diminuir a multiplicidade de um relacionamento. A figura 11, de (FURLAN, 1998), demonstra isso.

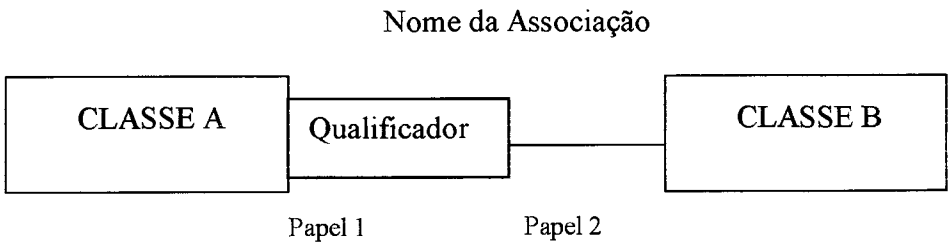


Figura 11 – Qualificador

4.4.5. Agregação/Composição

Para JAIME (2002), uma agregação é uma forma especial de associação utilizada para mostrar que um tipo de objeto é composto, pelo menos em parte, de outro, em relação de todo/parte

Os "objetos-parte" não podem ser destruídos por qualquer objeto diferente do objeto que o criou. Esse conceito se assemelha na análise estrutura ao conceito de pai e filho, onde o filho só pode ser criado ou destruído pelo pai. A figura de FURLAN (1998) abaixo demonstra esse conceito.

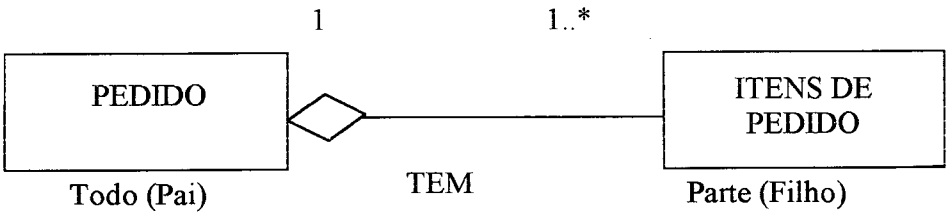


Figura 12- Agregação

O símbolo de agregação (◇) vazado indica uma agregação regular, também conhecida na UML por relacionamento *por referência*. Se ele esta cheio (◆), por *default*, significa uma forma forte de agregação conhecida por composição ou relacionamento *por valor*.

Composição é uma forma de agregação. Nessa forma de agregação, o objeto todo declara uma instância real do objeto para dentro do seu próprio corpo tornando o objeto parte fisicamente nele contido. A figura abaixo demonstra esse conceito.

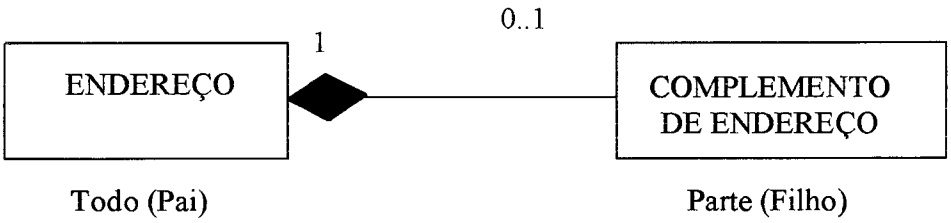


Figura 13 – Composição

4.4.6. Navegabilidade

As classes estão relacionadas entre si, através da maneira pela qual seus objetos correspondentes podem colaborar entre si para realizar uma funcionalidade mais complexa, sendo este relacionamento, por *default*, bidirecional. Mesmo levando isso em conta,

associações unidirecionais são importantes para especificação dos modelos de projeto ou implementação. A figura abaixo mostra o conceito.

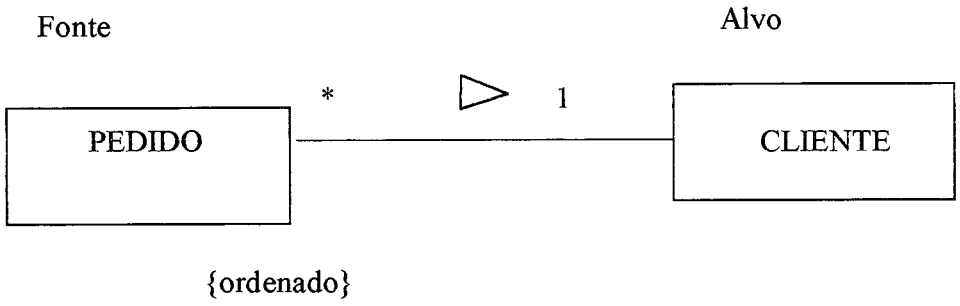


Figura 14 - Navegabilidade

4.4.7. Mutabilidade

Mutabilidade é aquilo que pode ser somado, apagado, removido ou que possa sofrer qualquer outro tipo de operação que cause modificação. Porém existe a propriedade {congelado} que indica a não possibilidade de que essas operações sejam efetuadas.

4.4.8. Visibilidade

Trata-se da mesma notação utilizada para visibilidade de atributos e operações. Especifica a visibilidade da associação que atravessa em direção ao nome em direção ao nome do papel determinado. Podem ser especificadas outras propriedades para pepéis de associações, mas não há qualquer sintaxe gráfica para eles. Para especificar tais propriedades utiliza-se sintaxe de restrição próxima ao fim do caminho de associação (uma seqüência de texto entre chaves) (FURLAN, 1998).

4.4.9. Generalização

Em UML, uma generalização é um relacionamento de taxionomia, entre um elemento mais geral e um elemento mais específico, que é completamente consistente com o primeiro elemento somando-o "informação adicional especializada".

Isso quer dizer que uma classe mais geral (superclasse) tem atributos, operações e associações comuns que são compartilhados por classes mais específicas (subclasses). Essas últimas, além herdam atributos e operações particulares ao elemento de especialização a que se referem.

No modelo conceitual, a generalização é mostrada como uma linha sólida do elemento mais específico para o mais geral, inclusive com um retângulo na ponta deste último.

Podem ser utilizadas restrições pré-definidas para indicar condições semânticas entre subclasses. Palavras-chaves são colocadas entre chaves, separadas por vírgula, próximas ao triângulo compartilhado; se vários caminhos compartilharem um único triangulo, ou então próximo a uma linha pontilhada que cruza as linhas de generalização envolvida.

As palavras-chaves são pré-definidas na UML. Elas podem ser:

a) **Sobreposição:** Subclasses derivadas de uma superclasse podem ocorrer simultaneamente para uma mesma ocorrência de superclasse;

b) **Disjunção:** Subclasses derivadas de uma superclasse podem ocorrer se são mutuamente exclusivas para uma mesma ocorrência da superclasse;

c) **Completo:** Todas as subclasses foram especificadas, se mostradas ou não. Nenhuma subclasse adicional é esperada;

d) **Incompleto:** Algumas subclasses foram especificadas, mas a lista, por não ser conhecida está incompleta.

Quando estamos falando de generalização cabe frisar que a noção da concepção de herança é muito importante. A herança é o mecanismo de reutilização de atributos e operações definidos em classes gerais por classes mais específicas podendo ser usada para expressar tanto generalização quanto associação.

Provavelmente herança é um dos aspectos mais interessantes da modelagem orientada a objetos, pois provê um meio para se construir componentes supostamente reutilizáveis, representando uma expressão de similaridade para simplificar a definição de classes com

características semelhantes. Em se falando de herança, ela pode se classificada como simples e múltipla.

Herança simples é aquela onde uma subclasse herda estrutura e/ou comportamento de uma única superclasse. Já a herança múltipla e aquela cuja uma subclasse herda estrutura e/ou comportamento de mais de uma superclasse

Herança é uma relação entre classes de objetos e não uma relação entre instâncias das classes.

Uma subclasse herda atributos, operações e relacionamentos e pode adicionar novos atributos, operações e relacionamentos e redefinir operações herdadas, assim:

a) **Herança de Atributos:** A subclasse de Objetos herda atributos da superclasse.

b) **Herança de Operações:** A subclasse de Objetos herda registros, por exemplo, da superclasse.

c) **Herança de Relacionamentos:** As subclasses de Objetos herdam relacionamentos existentes entre a superclasse.

Exemplos sobre generalização com herança simples e herança múltipla são demonstrados nas figura de (FURLAN, 1998) 15 e 16.

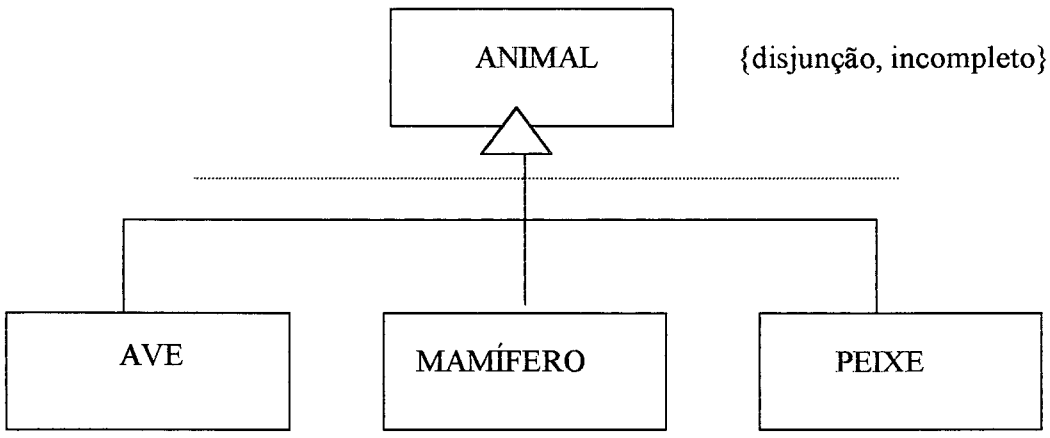


Figura 15 – Generalização com Herança Simples

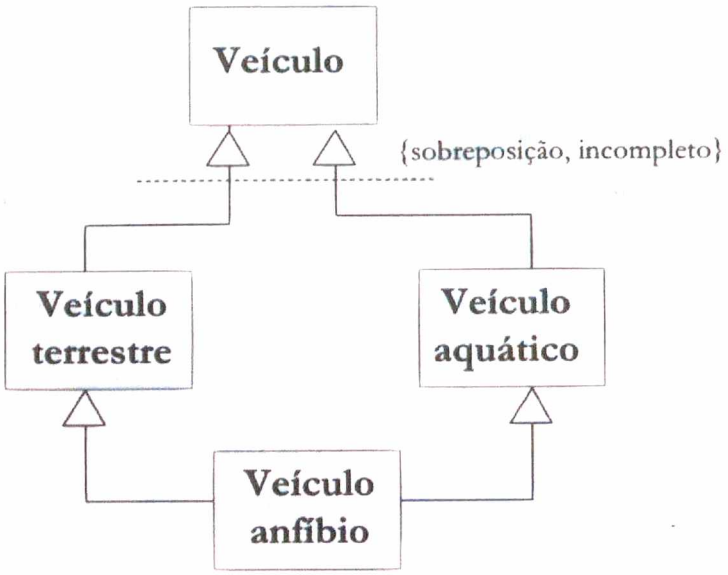


Figura 16 - Generalização com Herança Múltipla

4.5. Os Atributos do Modelo Conceitual da UML

Como já frisado, o modelo conceitual é a representação de coisas do mundo real e neste sentido, os atributos a ele relacionados não podem deixar de seguir essa norma e devem ser interpretados no contexto de entidades do mundo real.

Para (LARMAN, 2000), um atributo é um valor de dados lógicos de um objeto. Em um modelo conceitual, os atributos são aqueles para os quais os requisitos sugerem ou implicam uma necessidade de memorização de informações. Porém, é importante observar que algumas coisas não devem ser tratadas como atributos e sim como associações.

No modelo conceitual, conceitos são relacionados através de uma associação e não de um atributo. Na modelagem conceitual, os atributos deveriam ser valores de dados puros ou tipo de dados, ("*Data Types*") como diz a nomenclatura em UML, aqueles para os quais uma identidade única não faz sentido no contexto do modelo ou sistema (RUMBAUGH, 1991).

Porém, é de extrema importância distinguir (através de identidade única), entre duas instâncias separadas de duas pessoas, que, por exemplo, se chamam José Soares, mas que são pessoas diferentes e possuem dados de identificação pessoal também diferentes.

Quando falamos de modelo conceitual, é sempre útil ter em mente que é melhor que ele possua mais conceitos ao invés de conceitos muito genéricos. Por este motivo, quando em um conceito não cabem os conceitos de número, string, boolean, e data entre alguns outros é importante representar como um novo conceito.

Não existe algo que seja um único modelo correto. A maioria dos modelos são aproximações do domínio que o analista está tentando compreender. Um bom modelo conceitual captura as abstrações essenciais e as informações requeridas para compreender o domínio, no contexto dos requisitos presentes, auxiliando as pessoas na compreensão desse domínio, seus conceitos, sua terminologia e seus relacionamentos (LARMAN, 2000).

CAPÍTULO V

ERROS OBSERVADOS NA CONSTRUÇÃO DE MODELOS CONCEITUAIS

5.1. Introdução

Uma das maiores preocupações da análise de sistemas é com a compreensão do domínio do problema. Para conseguir a compreensão do domínio do problema, o analista na fase de entrevistas com o usuário, deve obter o máximo de informações possíveis sobre o domínio do problema e ter consciência de que as pessoas envolvidas com o sistema são importantes para se chegar à compreensão do domínio do problema. Um modelo conceitual ilustra os conceitos significativos (para modeladores) do domínio de um problema. Ele é um artefato muito importante a ser criado durante a análise orientada a objetos.

Erros de modelagem que possam ocorrer na fase de elaboração do modelo conceitual podem causar inúmeros problemas em fases futuras do desenvolvimento do projeto e conseqüentemente na implementação do software.

Para o desenvolvimento do problema de pesquisa desta dissertação foram analisados dezenove (19) trabalhos realizados por aprendizes de graduação e pós-graduação, das disciplinas de Análise e Projeto de Sistemas Computacionais e Sistemas Orientados a Objetos, e bibliografia específica da área (LARMAN, 2000), (LARMAN, 2002), (FURLAN, 1998), (MULLER, 1997), entre outros, o que não é um indicativo de final de pesquisa, mas sim um início, pois além dos erros apontados, podem existir outros que a qualquer momento podem ser incorporados ao modelo (proposta da dissertação).

Após uma apurada avaliação dos mesmos, detectou-se como sendo 14 (quatorze) os erros mais freqüentemente praticados pelos aprendizes. Os mesmo apareceram em elevada quantidade e em inúmeros trabalhos ao mesmo tempo.

5.2. Identificação dos Erros de Modelagem

Os erros de modelagem serão apresentados na forma de regras. Estas regras foram desenvolvidas baseadas na observação da quantidade de erros realizados por aprendizes e nos erros de modelagem que as referências bibliográficas apontaram.

Para tanto foi necessária a identificação dos erros, o desenvolvimento das regras, a identificação de qual é o momento apropriado para que esta regra seja aplicada, no texto denominado como MOMENTO DE APLICAÇÃO. Verificou-se o grau de certeza de aplicabilidade que a regra possui, no texto denominado como GRAU DE CERTEZA DA REGRA, que variou entre "ERRO", quando o grau de certeza da regra era máximo, "ADVERTÊNCIA", quando o grau de certeza da regra era de muito alto até médio e "SUGESTÃO", quando o grau de certeza da regra era mínimo. Além disso, levantou-se alguns questionamentos que ainda pairaram sobre algumas regras.

Os erros de modelagem foram classificados conforme a ordem em que foram aparecendo e não por grau de importância e/ou quantidade de ocorrências. A tabela 01 informa a quantidade de ocorrência dos erros de modelagem conceitual nos trabalhos analisados.

Erros de Modelagem	Número de Ocorrências	Proporção com Relação ao Total
Classe_Plural – Advertência	3	15,78%
Classe_Coletiva – Advertência	3	15,78%
Direcionamentos_Incorretos – Erro	2	10,52%
Falta_Atributos – Erro ou Advertência	5	26,31%
Atributos_Repetidos – Sugestão	1	5,26%
Associações_Incorretas – Sugestão	2	10,52%
Atributo_Que_É_Referência – Sugestão ou Erro	2	10,52%
Sem_Multiplicidade – Advertência	1	5,26%
NomeAssociação_Incorreto – Erro	1	5,26%
Multiplicidade_Incorreta – Sugestão	2	10,52%
Classes_Com_Métodos – Erro	1	5,26%
Atributos_Com_Códigos – Advertência	2	10,52%
Classes_De_Software – Erro	3	15,78%
Associações_Sem_Nome – Sugestão	1	5,26%

Tabela 01 – Erros de Modelagem, Número de Ocorrências e Proporção em Relação ao Total.

5.2.1. Erro 1: *Criação de uma classe que está no plural.*

A regra 1 relaciona-se com a criação de uma classe que na verdade representa um conjunto de objetos. Freqüentemente os estudantes esquecem que conjuntos de entidades são representados através de associações com multiplicidade maior do que 1, e criam conceitos que criam conceitos que representam, por exemplo, "conjunto de clientes", "acervo", etc. A Figura 17 representa este erro.

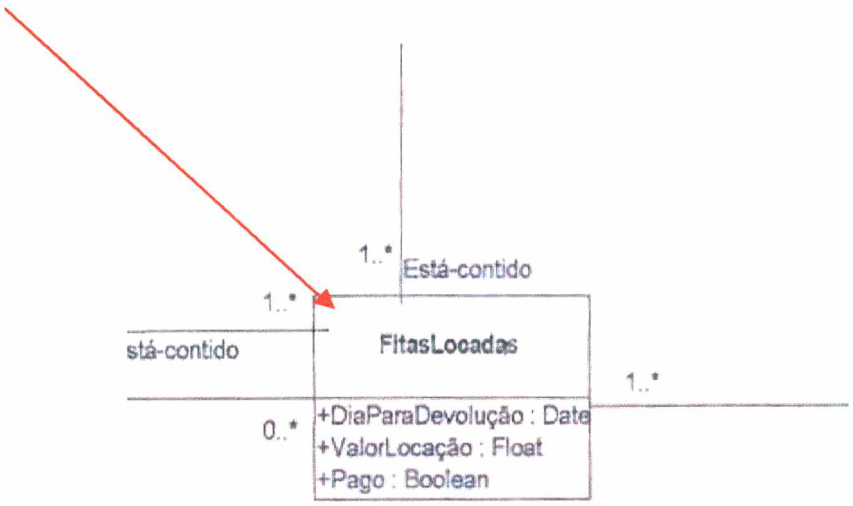


Figura 17 – Classe_Plural (Colocou nomes de Classes no Plural)

Uma maneira de identificar este tipo de erro é verificar se no conjunto de entidades do modelo conceitual existem nomes no plural. Nestes casos, normalmente, trata-se de um erro de modelagem como o descrito acima. Pode-se sugerir ao aprendiz que revise a necessidade

de existência desta entidade uma vez que o conjunto de entidade escrito pode ser representado como uma associação com multiplicidade maior que 1.

Regra 1: Classe_Plural

SE o nome da classe está no plural.

ENTÃO informar ao aprendiz que conjunto de elementos são denotados na multiplicidade das associações e que não se criam classes para denotar estes conjuntos, a não ser que estes conjuntos possuam atributos próprios que não podem ser derivados dos atributos de seus elementos.

Momento da aplicação: No momento em que a classe for associada à outra, no momento que outra classe ou associação for editada ou no momento de salvar.

Grau de certeza da regra: "ADVERTÊNCIA", pois sempre que existirem classes que estão no plural, como por exemplo "máquinas de amolar", "clientes", entre outros, existe grande probabilidade de erro.

5.2.2. Erro 2: *Criação de uma classe que representa um conjunto*

A regra 2 relaciona-se com a criação de uma classe que na verdade representa um conjunto de objetos. Freqüentemente os estudantes esquecem que conjuntos de entidades são representados através de associações com multiplicidade maior do que 1, e criam conceitos que criam conceitos que representam, por exemplo, "conjunto de clientes", "acervo", etc. A Figura 18 representa este erro.

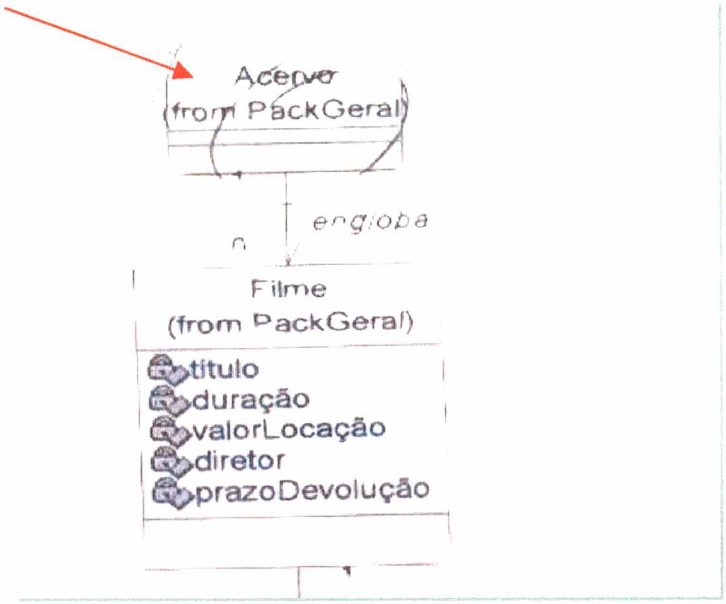


Figura 18 – Classe_Coletiva (Usou substantivo coletivo para nomear a Classe)

Uma maneira de identificar este tipo de erro e que normalmente produz bons resultados é verificar se no conjunto de entidades do modelo conceitual existem nomes que representam substantivos coletivos, nestes casos normalmente trata-se de um erro de modelagem como o descrito acima. Pode-se sugerir ao aprendiz que revise a necessidade de existência desta entidade uma vez que o conjunto de entidade escrito pode ser representado como uma associação com multiplicidade maior que 1.

Regra 2: Classe_Coletiva

SE o nome da classe é um substantivo coletivo

ENTÃO informar ao aprendiz que conjunto de elementos são denotados na multiplicidade das associações e que não se criam classes para denotar estes conjuntos, a não ser que estes conjuntos possuam atributos próprios que não podem ser derivados dos atributos de seus elementos.

Momento da aplicação: No momento em que a classe for associada à outra, no momento que outra classe ou associação for editada ou no momento de salvar.

Grau de certeza da regra: "ADVERTÊNCIA", pois sempre que existirem classes que estão no plural ou que possuem nome que é um substantivo coletivo, como por exemplo "acervo", entre outros, existe grande probabilidade de erro.

5.2.3. Erro 3: *Colocar direcionamento nas associações*

A regra 3 trata dos erros de modelagem que ocorrem em função da colocação de direcionamentos nas associações do modelo conceitual. A ocorrência deste tipo de erro pode ser observada na Figura 19.

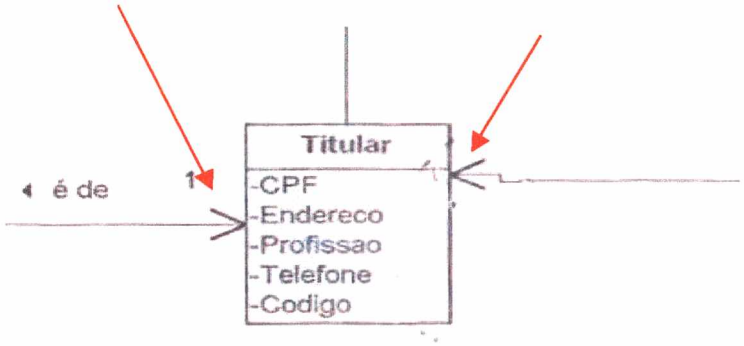


Figura 19 - Direcionamentos_Incorretos (Colocou Direcionamentos no Modelo Conceitual)

Uma boa forma de identificar estes erros pode ser na hora da criação da associação, informando ao aprendiz que o direcionamento das associações só poderá ser formulado de maneira correta em fases posteriores.

Regra 3: Direcionamentos_Incorretos

SE a associação possui direcionamento.

ENTÃO informar ao aprendiz que não se coloca direcionamentos no modelo conceitual, visto que somente com as informações dos Diagramas de Colaboração da fase de projeto é que se poderá saber como direcionar as associações de maneira apropriada.

Momento da aplicação: No momento da criação do direcionamento.

Grau de certeza da regra: "ERRO", pois com 100% de certeza os direcionamentos são colocados nos Diagramas de Colaboração de um projeto e não no modelo conceitual.

5.2.4. Erro 4: *Criação de classes sem atributos ou com um único atributo*

A regra 4 se relaciona com as classes que não possuem atributos ou com as classes que possuem apenas um atributo que corresponde ao próprio nome da instância.

Muitos aprendizes, na ânsia de desenvolver seus modelos conceituais, seguem regras que foram compreendidas de forma incorreta, como a regra definida por (LARMAN, 2000), que afirma que o número de classes de um modelo conceitual não é importante, mas sim não deixar classes fora deste modelo.

Em algumas situações os aprendizes criam várias classes e não fazem as associações necessárias para que estas classes pudessem se relacionar com outras classes de forma mais correta. A figura 20 demonstra este erro.

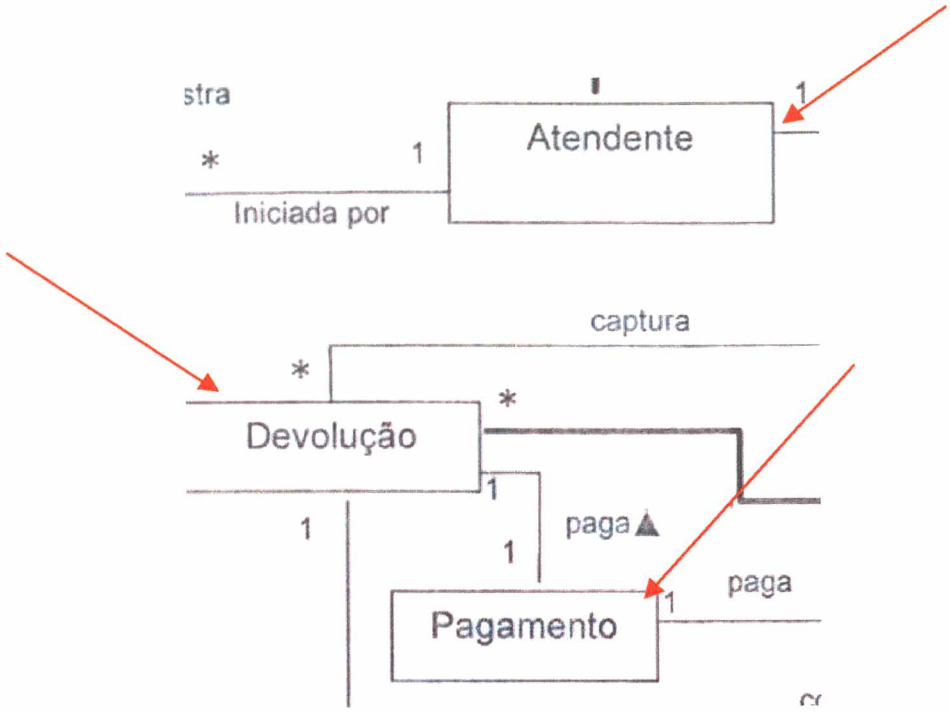


Figura 20 – Falta_Atributos (Criou Classes com um único Atributo ou sem Atributos)

A figura acima mostra algumas situações de erro onde as classes não possuem atributos. As classes sem atributos, neste caso, devem ser observadas com atenção, pois se a classe sem atributos se associar a duas ou mais classes ela pode funcionar apenas como elo de ligação. Nesta situação a regra possui um grau de certeza baixo e deve ser observada antes de qualquer alteração.

Já os casos em que a classe está ligada a apenas uma outra classe denotam a certeza da existência do erro, pois classes sem atributos que estão ligadas a apenas uma outra classe podem se tornar uma atributo da classe a que está ligada, então a regra possui grau máximo de certeza.

Uma boa maneira de identificar a existência destes problemas é alertar o aprendiz para que observe a regra criada para esta situação.

Regra 4: Falta_Atributos

SE a classe não possui atributos ou possui um único atributo e a classe está ligada a apenas uma outra classe.

ENTÃO informar o aprendiz que, com raras exceções, as classes devem possuir um atributo para conter informações. Apenas no modelo de projeto se admite classes de serviço, que por definição podem não possui nenhum atributo. Pode-se sugerir ao aprendiz que implemente a informação da classe como um atributo simbólico na outra classe a que ela faz referência.

Momento da aplicação: No momento de salvar,

Grau de certeza da regra: Depende da habilidade de identificar os casos, pode ir de uma "ADVERTÊNCIA" até um "ERRO", erro quando a classe for criada sem atributos e com uma única associação. Já a advertência se a classe quando a classe for criada sem atributos mas com mais de uma associação.

5.2.5. Erro 5: *Criação de nomes de atributos repetidos em duas classes ou mais classes*

O erro 5 diz respeito a criação de atributos repetidos em mais que uma classe. A Figura 21 aponta a existência deste erro.

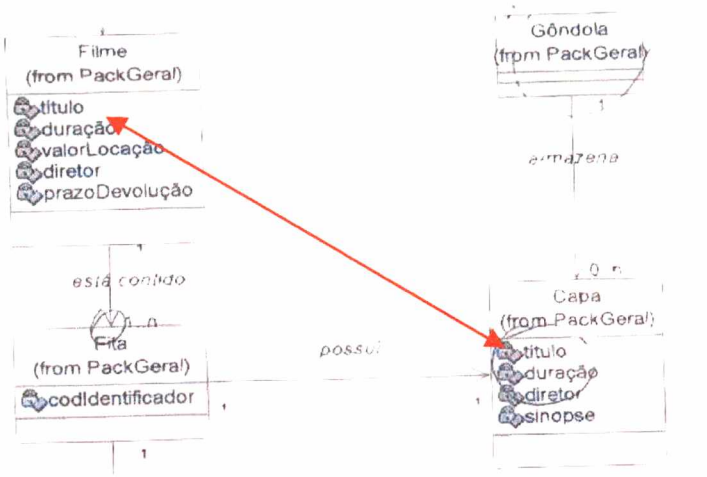


Figura 21 – Atributos_Repetidos (Criou Atributos Repetidos em duas ou mais Classes)

Em alguns casos, dependendo da proximidade entre as classes estes atributos que estão repetidos poderiam ser apenas ligados por associações para compartilhar informações.

A definição da proximidade entre as classes que possuem atributos repetidos é imprescindível para a identificação da possibilidade do uso de associações e para a informação ao aprendiz da possibilidade de aplicação da regra definida abaixo.

Regra 5: Atributos_Repetidos

SE duas classes possuem os mesmos atributos.

ENTÃO informar ao aprendiz que os atributos podem estar sendo repetidos desnecessariamente e que uma associação entre as duas classes poderia permitir o acesso às informações ou mesmo a aplicação de herança na associação.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: "SUGESTÃO", pois depende muito da capacidade do aprendiz de identificar a necessidade de redundância do atributo, ou seja, quanto mais perto as classes que possuem os mesmos atributos estiverem, maior se torna a certeza da regra. Quanto mais longe elas estiverem o grau de certeza da regra vai diminuindo.

5.2.6. Erro 6: *Criação de associações que são na realidade operações (transformações de dados)*

Esta regra diz respeito a associações que possuem como nome verbos que denotam transformação de informações. A Figura 22 demonstra este erro, pois o verbo distribuído por, do modelo conceitual do jogo Banco Imobiliário, denota uma ação em que existe a transformação de dados.

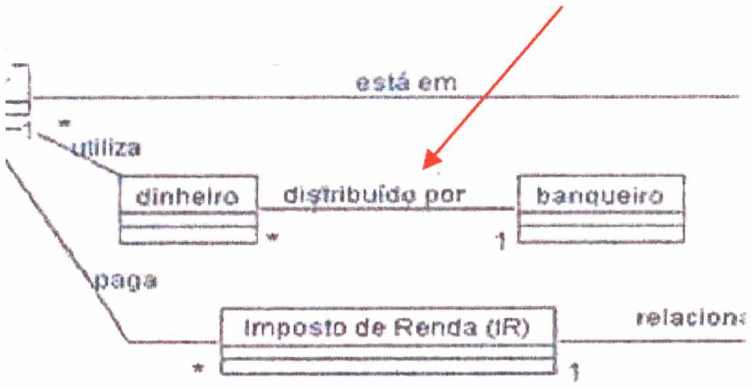


Figura 22 – Associações_Incorretas (Nome da Associação denota transformação de dados)

Regra 6: Associações_Incorretas

SE o nome da associação é um verbo que denota transformação de informação.

ENTÃO informe o aprendiz que as associações devem representar informações estáticas sobre a estrutura de conceitos e não as transformações dinâmicas.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: "SUGESTÃO", pois depende muito da capacidade de identificar os verbos que denotam transformação de informações.

5.2.7. Erro 7: *Criação de atributos que são referência para uma classe ou atributos que são nomes de uma classe do modelo conceitual.*

O erro 7 relaciona-se com a existência de atributos que fazem referência para outras classes ou que até mesmo possuem o nome de outra classe do modelo conceitual. Em muitos casos, os aprendizes esquecem que as referências feitas em determinadas classes a outras classes podem ser realizadas através de associações entre classes.

Regra 7: Atributo_Que_É_Referência

SE o nome do atributo indica que ele é uma referência para outra classe ou atributos possuem o mesmo nome de uma classe do modelo conceitual.

ENTÃO informar ao aprendiz que ele deve usar uma associação para ligar conceitos e não atributos.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: Dependendo da capacidade de identificar os prefixos que indicam a existência de referências para outras classes pode ir de "SUGESTÃO" até

'ERRO". Ocorre esta variação pois o algoritmo definido pode ou não ser capaz de identificar corretamente os todos os prefixos existentes.

5.2.8. Erro 8: *Não especificou multiplicidade nas associações*

Este erro diz respeito à não existência de notação de multiplicidade em diversas associações de um mesmo modelo conceitual. A Figura 23 indica este erro.

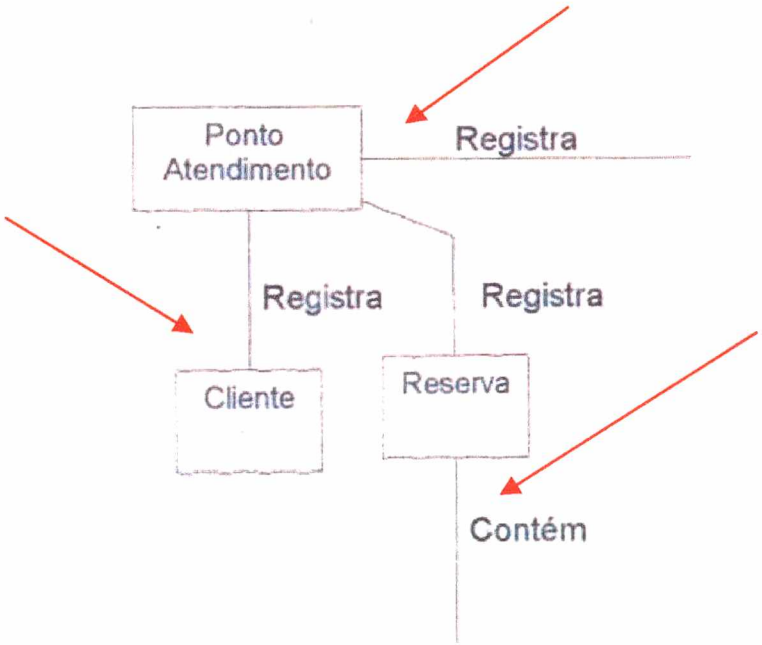


Figura 23 – Sem_Multiplicidade (Não colocou notação de Multiplicidade nas Associações)

Este erro pode ser facilmente solucionado, informando ao aprendiz que as associações sem notação de multiplicidade são sempre consideradas como sendo de 1 para 1.

Regra 8: Sem_Multiplicidade

SE várias associações do modelo não possuem notação de multiplicidade.

ENTÃO informar que as associações sem notação de multiplicidade são consideradas como sendo de 1 para 1.

Momento da aplicação: No momento de salvar, porém há que se ter o cuidado pois existem fases em que ainda não há interesse do aprendiz em coloca-las no modelo conceitual, deixando-as para a final da modelagem conceitual.

Grau de certeza da regra: "ADVERTÊNCIA", pois devem existir notações de multiplicidade no modelo conceitual.

5.2.9. Erro 9: *Especificou-se relação "é" ou "é-um" como associação simples*

Esta regra relaciona-se com a nomenclatura dada a associação. Esta nomenclatura, em alguns casos, tende a parecer uma forma de herança (já definido no Capítulo II e Capítulo IV) e herança em UML possui um símbolo específico. A Figura 24 aponta para este erro, que, pode ser resolvido com a aplicação da regra definida abaixo.

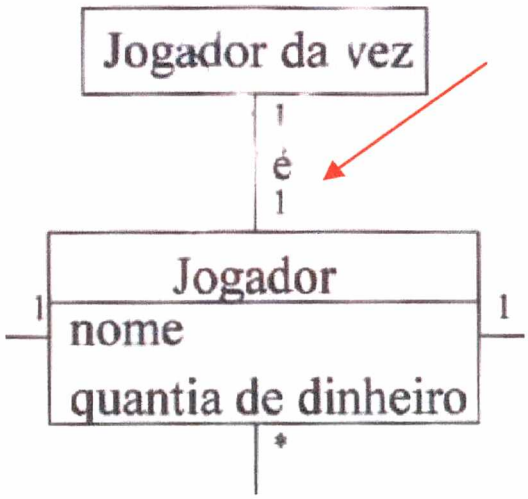


Figura 24 – NomeAssociação_Incorreto (Trata-se provavelmente de Herança)

Regra 9: NomeAssociação_Incorreto

SE uma associação é rotulada com "é" ou "é-um" ou similar.

ENTÃO informar ao aprendiz que se trata provavelmente de herança, ou que o nome da associação é inadequada.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: "ERRO", pois herança não deve ser rotulada como nome de associações e sim da maneira adequada, definida para o uso de herança.

5.2.10. Erro 10*: Especificou-se multiplicidade " um-para-um"*

Este erro diz respeito à existência de notação de multiplicidade 1 para 1 em diversas associações de um mesmo modelo conceitual. A Figura 25 aponta para a existência deste erro. Este erro pode ser facilmente solucionado, informando ao aprendiz que as associações com notação de multiplicidade 1 para 1 são difíceis de encontrar e que pode ser aplicada a regra abaixo.

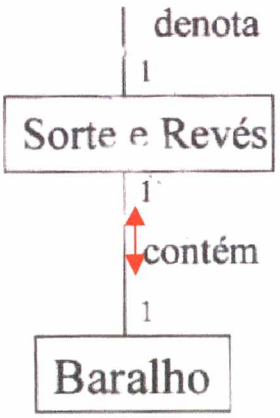


Figura 25 – Multiplicidade_Incorreta (Criou Multiplicidade de 1 para 1)

Regra 10: Multiplicidade_Incorreta

SE uma associação é rotulada como 1 para 1.

ENTÃO informar ao aprendiz que relações de multiplicidade de 1 para 1 sugerem que as classes envolvidas ba associação sejam agrupadas.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: "SUGESTÃO", pois, podem existir associações de 1 para 1. Porém, como estas associações são raras, quanto maior o número de vezes que este tipo de associação ocorrer em um mesmo modelo conceitual maior a probabilidade de se tratar de um erro.

5.2.11. Erro 11: *Colocou-se métodos nas classes*

Em algumas situações os aprendizes colocam métodos que são, segundo LARMAN (2000), operações que implementam algo, como, por exemplo, imprimir. Sabe-se, porém, que somente elementos do mundo real existem no modelo conceitual e não entidades do software.

Neste sentido, quando um aprendiz colocar um método em um modelo conceitual, pode-se aplicar a regra que informa ao aprendiz que métodos não pertencem aos modelos conceituais e sim aos Diagramas de Colaboração. A Figura 26 aponta para este erro.

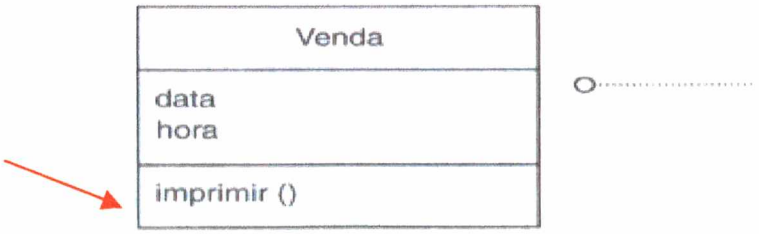


Figura 26 – Classes_Com_Métodos (Colocou Métodos nas Classes)

Regra 11: Classes_Com_Métodos

SE uma classe possui métodos.

ENTÃO informar ao aprendiz que o modelo conceitual não deve apresentar métodos, ficando os mesmos para serem feitos no Diagrama de Classes do Projeto, após a construção dos Diagramas de Colaboração.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: "ERRO", pois com 100% de certeza os métodos não são aplicados no modelo conceitual.

5.2.12. Erro 12: *Colocou-se códigos como atributos em uma ou mais classes*

A regra se relaciona à existência de códigos não existentes no mundo real, como, por exemplo, Código, cd, cód, etc. e não com siglas, como CPF, CGC, entre outros. A Figura 27 aponta para este erro.

Figura 27 – Atributo_Com_Código (Colocou como Atributos no Modelo Conceitual)

Regra 12: Atributos_Com_Códigos

SE uma classe apresenta um atributo com o nome "código" ou semelhante.

ENTÃO indicar ao aprendiz que no modelo conceitual só se usa códigos que existam no mundo real, como por exemplo, CPF. Códigos de identificação de objetos são restritos ao modelo de implementação, não devendo aparecer no modelo conceitual.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: "ADVERTÊNCIA", pois no modelo conceitual podem existir códigos que pertencem ao mundo real e são representados por palavras como "código".

5.2.13. Erro 13: *Colocou-se classes que são artefatos de software (banco de dados, relatórios, dispositivos, arrays, etc.)*

Esta regra verifica e informa a existência de erros quanto artefatos de software são encontrados no modelo conceitual. A Figura 28 demonstra este erro.

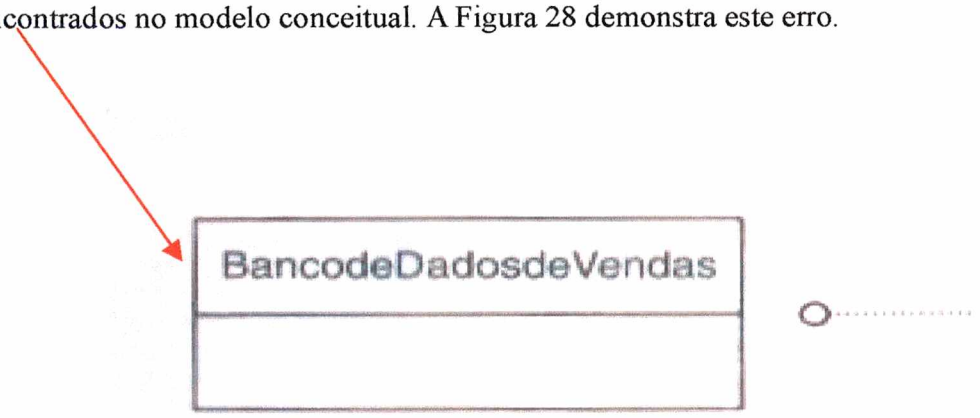


Figura 28 – Classes_Com_Software (Colocou Classe como Software)

Regra 13: Classes_Com_Software

SE uma classe apresenta artefatos de software.

ENTÃO informar ao aprendiz que o modelo conceitual trata de elementos do mundo real e não elementos de software.

Momento da aplicação: No momento em que a classe for associada à outra, ou no momento de salvar.

Grau de certeza da regra: "ERRO", pois software não está associado ao modelo conceitual. Porém tem-se que observar, pois existem situações especiais onde o domínio modelado é de software.

5.2.14. Erro 14: *Criou associações sem nomes*

Esta regra verifica e informa a não existência de nomes nas associações do modelo conceitual. A Figura 29 demonstra este erro.

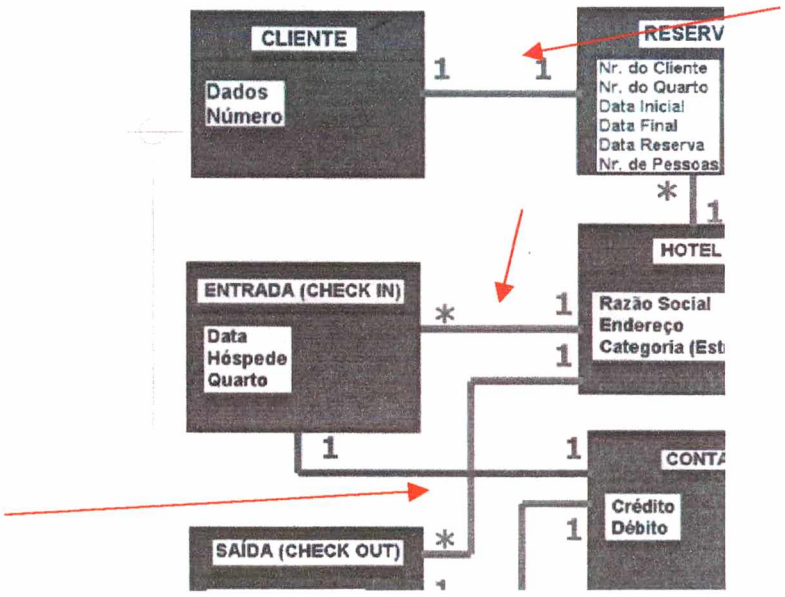


Figura 29 – Associações_Sem_Nome

Regra 14: Associações_Sem_Nome

SE uma associação se apresenta sem nome.

ENTÃO informar ao aprendiz que todas as associações devem possuir nome.

Momento da aplicação: No momento de salvar.

Grau de certeza da regra: "SUGESTÃO", pois toda associação entre classes possui um nome. Há que se observar que dependendo do estágio da modelagem, pode ser que o usuário ainda não queira definir o nome das associações.

CAPITULO VI

RESOLUÇÃO INFORMATIZADA PARA AS REGRAS DEFINIDAS

6.1. Introdução

Para os erros encontrados e definidos como os mais freqüentemente observados nos modelos conceituais, foram criadas regras que possibilitassem a resolução deste problema.

Para que estas regras pudessem ser aplicadas como verificadoras de erros cometidos por aprendizes, foram definidas operações para a definição do domínio do problema baseadas na notação Backus-Naur Form (BNF), que é uma meta-linguagem usada para descrever sintaxe de linguagens. A especificação consiste em um terminal do lado esquerdo, e um conjunto de uma ou mais produções do lado direito separados pelo símbolo "::=" (NETO, 1987).

6.2. Formalização dos Dados

6.2.1. Operações de Definição para o Domínio do Problema

A BNF Estendida (BNF Extended) é equivalente à BNF, mas permite uma especificação mais compacta da sintaxe de uma linguagem de programação.

Os meta símbolos da notação BNF Estendida utilizados no desenvolvimento do modelo são:

1. Concatenação: Usa-se apenas a justaposição de símbolos para representar concatenação
2. Não-Terminal: Representado por um nome delimitado pelos símbolos "<" e ">" Ex: <x>

3. Terminal: Uma seqüência de caracteres alfa-numéricos representando um nome.
4. Lista: Representado por * (asterisco) colocado após o símbolo que deseja repetir. Para repetir uma seqüência de símbolos, usam-se parênteses para delimitar a abrangência do operador de lista.
5. Mapeamento: Representado por →.
6. União: Representado por U.
7. Definido como: Representado por ::=
8. Separador de Produções: Representado por | (lê-se: "ou")
9. Literal: Uma palavra literalmente escrita se apresenta sempre entre aspas " " ou simplesmente escrita.
10. Elementos Simples: São representados entre parênteses ( ). Usa-se parênteses para delimitar a aplicação dos operadores.
11. Elementos Opcionais: São representados entre colchetes [ ].
12. Conjunto Vazio: Representado pelo símbolo grego épsilom ε.
13. Termos Booleanos: Representados como literais, "T"| "F".

6.2.2. Definição dos Termos Conforme Formalização dos Dados

6.2.2.1. Atributos

Atributos são nomes com tipos, como por exemplo, idade: Numeral, então conforme formalização:

<Atributo>::=<NomeAtributo>: <TipoAtributo>

6.2.2.2. Nome dos Atributos

O termo NomeAtributo diz respeito ao nome que os atributos possuem, por exemplo, Nome, CPF, etc. A String definida é uma string qualquer de palavras.

<Nome Atributo >::=<String>

6.2.2.3. Tipo dos Atributos

O termo TipoAtributo diz respeito ao nome que os tipo dos atributos possuem, por exemplo, inteiro, boolean, etc. A String definida é uma string qualquer de palavras.

<TipoAtributo >::=<String>

6.2.2.4. Conceitos

Conceitos são nomes associados a uma lista de atributos e métodos (operações), ligadas à classe e seus atributos, como por exemplo Automóvel(ano:Numeral), então

<Conceito>::=<NomeClasse>(<Atributo>*) (<Método>*)

6.2.2.5. Associações

Definidas como as ligações entre duas classes. Possuem o nome da classe a que está ligado, seguido de uma notação de multiplicidade, do nome que a associação possui, de outra notação de multiplicidade, do nome da outra classe a que está ligada e de uma direção de leitura. É a ligação entre as classes. Exemplo: Pessoa/1/possui/*/Automóvel/-.

Também é importante observar que herança é um atributo da linguagem, assim como a agregação. Então, com relação ao nome da associação, sempre se deve ter atenção para o fato de se estar representando associações normais, com herança, com agregação, ainda onde existam qualificadores ou ordenação, além dos casos que o modelo não prevê.

É bom também, observar que uma restrição semântica para que uma especificação de multiplicidade esteja bem formada é que os números que nela aparecem sejam sempre crescentes da esquerda para a direita.

<Associação>::=<NomeClasse>/<CasosEspeciais>|<CasosEspeciais_1>/<Multiplicidade>/NomeAssociação>/<Multiplicidade>/<NomeClasse>/<Direção>

6.2.2.6. Casos Especiais

São os casos em que há a existência de herança, agregação, qualificadores e ordenação nas associações.

<CasosEspeciais>::="Herança"|"Agregação"|"Qualificador"|"Ordenação"| ε

6.2.2.7. Casos Especiais_1

São os casos que pode também ocorrer casos com agregação e qualificador como também agregação e ordenação e ainda agregação com qualificador e ordenador.

<CasosEspeciais_1>::="Agregação e Qualificador"| "Agregação e Ordenação"|"Agregação e Qualificador e Ordenador"| ε

6.2.2.8.Multiplicidade

Representa a anotação de multiplicidade que a associação possui.

<Multiplicidade>::=*|<Num>|<Num><Complemento Mult>

6.2.2.9. Complemento Mult

Representa a anotação que complementa o valor representante da multiplicidade.

<Complemento Mult::= ε | , **<Num><Complemento Mult>**| - **<Num><ComplementoMult>**| *; onde Num representa o conjunto dos números naturais {0, 1, 2, 3...}

6.2.2.10. Direção

Representa o sentido das associações (unidirecional para a direita,unidirecional para a esquerda, bidirecional e sem direção).

<Direção>::= "→"|"←"|"↔"|"—"

6.2.2.11. Modelo

É um conjunto de classes, com seus atributos e associações.

<Modelo>::=<Conceito>*U<Associação>*

6.3. Funções Definidas para o Desenvolvimento do Modelo

Para o desenvolvimento dos algoritmos necessários para a criação do modelo proposto será necessária a definição de funções que ajudarão na formulação dos mesmos. Abaixo segue as funções necessárias para sua formulação.

6.3.1. Plural

É uma função que aplicada a uma string "x" (que no caso é um termo) retorna o valor "True" se "x" representar uma sentença que denota um elemento no plural. A função retorna "False" caso contrário. Neste caso, é importante lembrar que a sentença "x" está no singular ou que está mal formada.

Plural(x) = True se x está no plural

Plural(x) = False se x está no singular ou está mal formada.

6.3.2. Vazio

Uma classe sem nenhum atributo é denominada como vazio é uma função que aplicada a uma string "x" (que no caso é um conjunto) retorna o valor "True" se "x" representar uma sentença que denota um conjunto vazio, ou seja, sem nenhum atributo. A função retorna "False" caso a classe possuir um ou mais atributos.

Vazio(x) = True se x for um conjunto vazio (nenhum atributo)

Vazio(x) = False se x é um conjunto que possui atributos

6.3.3. Único

Uma classe com um único atributo foi denominada como único. É uma função que aplicada a uma string "x" (que no caso é um conjunto de atributos) retorna o valor "True" se "x" representar uma sentença que denota um conjunto com um único atributo. A função retorna "False" caso a classe possuir mais atributos.

Único(x) = True se x for um conjunto com um único atributo

Único(x) = False se x é um conjunto que possui mais que um atributo

6.3.4.Trivial

O termo trivial foi definido como a união entre uma classe sem nenhum atributo e uma classe com apenas um atributo. É uma função que aplicada a uma string "x" (que no caso é um conjunto) retorna o valor "True" se "x" representar uma sentença que denota um conjunto que, na união entre um conjunto vazio e outro com um único elemento, na pior das hipóteses, apresentar um único elemento, ou seja, com um único atributo. A função retorna "False" caso a classe possuir mais atributos.

Trivial(x) = True se x for um conjunto que da união entre um conjunto vazio e um conjunto com um único elemento forma um conjunto vazio ou com único elemento.

Trivial(x) = False se x é um conjunto que da união possui mais que um atributo

6.3.5. VerboDeTransformação

O termo verbo de transformação indica os verbos que denotam transformação de informações. É uma função que aplicada a uma string "x" (que no caso é um termo) retorna o valor "True" se "x" representar uma sentença que denota um termo que no caso, é um verbo que denota transformações de informações. A função retorna "False" caso a contrário.

VerboDeTransformação(x) = True se x for um termo que denota transformações de informações

VerbodeTransformação(x) = False se x for um termo que não possui denotação de transformação de informações

6.3.6. Método

Um método representa uma ação incorreta realizada no Modelo Conceitual, pois um método não representa coisas do mundo real e sim componentes que são utilizados nos Diagramas de Colaboração. É uma função que aplicada a uma string "x" (que no caso é um termo) retorna o valor "True" se "x" representar uma sentença que possui um método. A função retorna "False" caso a contrário.

Método(x) = True se x for um termo que apresenta métodos nas classes

Método(x) = False se x é não possuir métodos

6.3.7. Coletivo

O termo coletivo indica os nomes que se subentende ser substantivos coletivos. É uma função que aplicada a uma string "x" (que no caso é um termo) retorna o valor "True" se "x" representar uma sentença que é uma palavra que representa substantivos coletivos. A função retorna "False" caso a contrário.

Coletivo(x) = True se x for um termo que representa palavras que são substantivos coletivos

Coletivo(x) = False se x é não representar estas palavras

6.3.8. Código

O termo código indica as palavras que não podem ser usadas como nome de atributos no modelo conceitual, pois no mesmo só se utiliza códigos que existam no mundo real. Ex: CPF, CGC, etc. É uma função que aplicada a uma string "x" (que no caso é um termo) retorna o valor "True" se "x" representar uma sentença que é uma palavra que representa a presença de um código que não pertence ao mundo real. A função retorna "False" caso contrário.

Código(x) = True se x for um termo que representa palavras que representam a presença de códigos

Código(x) = False se x não representar estas palavras

6.3.9. Software

Esta expressão indica quais são as palavras que não podem aparecer no modelo conceitual, pois representam dispositivos de software e não pertencem ao mundo real. É uma função que aplicada a uma string "x" (que no caso é um termo) retorna o valor "True" se "x" representar uma sentença que possui um termo que representa um dispositivo de

software. A função retorna "False" caso o nome da classe não for um dispositivo de software.

Software(x) = True se x for um termo que representa dispositivos de software

Software(x) = False se x for um termo que não representa dispositivos de software

6.3.10. Referência

Esta expressão identifica as palavras que fazem referência a uma classe dentro de outra. É uma função que aplicada a uma string "x" (que no caso é um termo) retorna o valor "True" se "x" representar uma sentença que possui uma referência à outra classe. A função retorna "False" caso contrário.

Referência(x) = True se x for um termo que faz referência a uma classe dentro de outra classe

Referência(x) = False se x for um termo que não faz este tipo de referência

6.3.11. Compara

Esta expressão compara todos os elementos de um Vetor V com eles próprios, menos com ele mesmo. É uma função que aplicada a uma string "x" (que no caso é um termo comparado com outros termos) retorna o valor "True" se "x" representar uma sentença que possui o mesmo valor que outro qualquer do vetor. A função retorna "False" é aplicada caso após a comparação com todos os elementos do vetor nenhum nome for igual.

Compara(x) = True se x for um termo que tiver repetições no vetor

Compara(x) = False caso contrário

6.4. Implementação em Software para a Validação da Proposta

As 02 (duas) regras escolhidas para a implementação foram: Classes_Com_Métodos e Associações_Sem_Nome. Estas regras foram implementadas na linguagem de programação C, versão BC31 da Borland. A implementação pode ser observada no ANEXO 02.

Os modelos conceituais que são construídos com o uso da ferramenta de modelagem Rational Rose, da Rational, podem ser transformados, com o uso de uma das opções disponíveis na ferramenta, em um código fonte do modelo conceitual construído.

Utilizando esta possibilidade, o software desenvolvido para a validação faz a leitura do código fonte do modelo conceitual construído, que é transformado para o formato txt.. Durante esta leitura ele identifica a existência de métodos na modelagem e informa seus nomes e que eles não podem estar presentes na modelagem conceitual, pois eles são elementos de software e no modelo conceitual somente são permitidos elementos do mundo real.

Após observar a existência de métodos ele faz novamente a leitura do código fonte gerado e identifica a não existência de nomes nas associações, dando como retorno a informação de quantas associações estão sem nomes e lembrando que todas as associações devem possuir nomes.

CAPÍTULO VII

VALIDAÇÃO DO MODELO PROPOSTO

A validação do modelo proposto foi realizada em duas fases. Numa primeira fase, foi realizada manualmente. Através desta atividade foram diagnosticados os erros definidos e aplicaram-se as regras desenvolvidas anteriormente.

Pegou-se como amostra para a realização da validação 11 (onze) trabalhos de aprendizes da disciplina de Análise e Projeto de Sistemas. Estes trabalhos pertencem a outra turma de alunos e não são os mesmos trabalhos utilizados para a identificação dos erros avaliados.

Para cada um desses dos 11 (onze) trabalhos da amostra foram aplicados os 14 (quatorze) algoritmos desenvolvidos como resolução para os erros identificados na proposta de dissertação.

Neste sentido, os 14 (quatorze) algoritmos foram aplicados individualmente e, sempre que ocorreu algum dos erros, o algoritmo o tratou como definido na regra.

Esta verificação resultou em 168 (cento e sessenta e oito) verificações algorítmicas e a proporção de erros detectada está mais bem definida na tabela abaixo:

Erros de Modelagem	Número de Ocorrências	Proporção com Relação ao Total
Classe_Plural - Advertência	5	45%
Classe_Coletiva - Advertência	5	45%
Direcionamentos_Incorretos – Erro	0	0%
Falta_Atributos – Erro ou Advertência	9	81,81%
Atributos_Repetidos – Sugestão	4	36,36%
Associações_Incorretas – Sugestão	2	18,18%
Atributo_Que_É_Referência – Sugestão ou Advertência	0	0%
Sem_Multiplicidade – Advertência	0	0%
NomeAssociação_Incorreto – Erro	4	36,36%
Multiplicidade_Incorreta - Sugestão	10	90,90%
Classes_Com_Métodos – Erro	0	0%

Atributos_Com_Códigos – Advertência	1	9,09%
Classes_De_Software – Erro	0	0%
Associações_Sem_Nome – Sugestão	1	9,09%

Tabela 2 – Erros Detectados na Validação do Modelo

Na segunda fase da validação, após a implementação das 2 (duas) regras, os 11 (onze) trabalhos usados como amostra para a validação foram refeitos, ou seja, o que antes era uma figura desenvolvida em diversas ferramentas, foi repassados para a ferramenta Rational Rose 98, da Rational, pois a mesma possui uma opção que gera o código fonte do modelo conceitual construído.

A partir deste código, o software implementado identifica a existência ou não de erros. Caso identificado um ou os dois erros implementados, o software retorna um aviso informando que possíveis erros foram encontrados. A figura 30 demonstra a presença de possíveis erros.

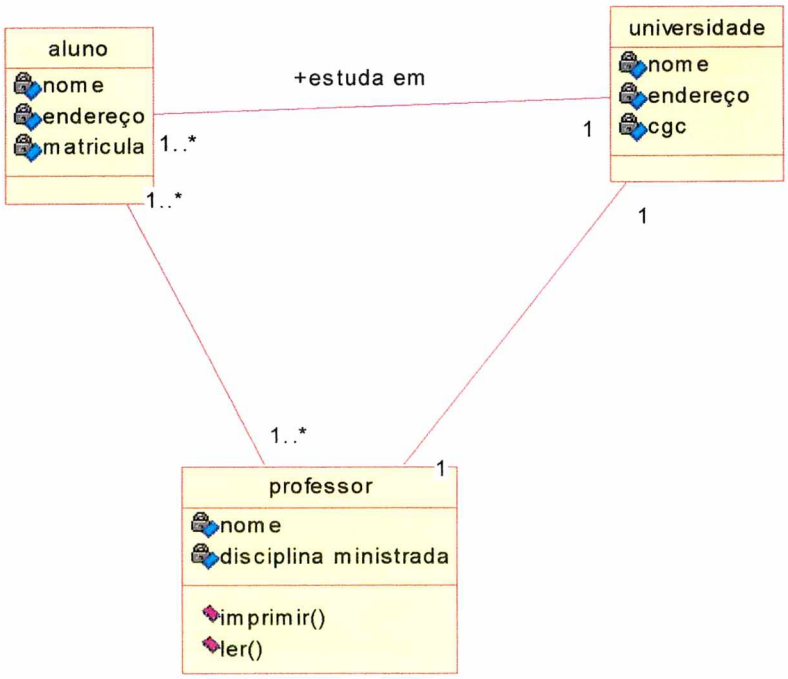


Figura 30 – Modelo Conceitual com Presença de Erros (Classe_Com_Métodos e Associações_Sem_Nome)

Para o modelo conceitual descrito acima, o Rational Rose gera o código fonte abaixo e aqui está representada somente a parte que interessa neste momento, que é gravado para a leitura do software desenvolvido para a validação em txt:

```
(object Petal
    version        42
    _written       "Rose 4.5.8163.3"
    charSet        0)

(object Design "Logical View"
    is_unit        TRUE
    is_loaded      TRUE
    defaults       (object defaults
     rightMargin   0.250000
     leftMargin    0.250000
     topMargin     0.250000
     bottomMargin  0.500000
     pageOverlap   0.250000
     clipIconLabels     TRUE
     autoResize    TRUE
     snapToGrid    TRUE
     gridX         16
     gridY         16
     defaultFont   (object Font
         size           10
         face           "Arial"
         bold           FALSE
         italics        FALSE
         underline      FALSE
         strike         FALSE
         color          0
         default_color TRUE)
     showMessageNum     3
     showClassOfObject TRUE
     notation       "Unified")
    root_usecase_package    (object Class_Category "Use Case View"
     quid           "32F8E3FE0029"
     exportControl "Public"
     global         TRUE
     logical_models     (list unit_reference_list)
     logical_presentations  (list unit_reference_list
         (object UseCaseDiagram "Main"
         quid           "32F8E3FE03CA"
         title          "Main"
         zoom           100
         max_height     28350
         max_width      21600
         origin_x       0
         origin_y       0
         items          (list diagram_item_list))))
```

```
root_category (object Class_Category "Logical View"
 quid          "32F8E3FE0028"
 exportControl "Public"
 global        TRUE
 subsystem     "Component View"
 quidu         "32F8E3FE002A"
 logical_models     (list unit_reference_list
     (object Class "aluno"
     quid          "3D47DF9D00C8"
     class_attributes  (list class_attribute_list
         (object ClassAttribute "nome"
         quid          "3D47DFB10320")
         (object ClassAttribute "endereço"
         quid          "3D47DFB403A2")
         (object ClassAttribute "matricula"
         quid          "3D47DFB702A8")))
     (object Class "universidade"
     quid          "3D47DFBD0226"
     class_attributes  (list class_attribute_list
         (object ClassAttribute "nome"
         quid          "3D47DFCA0190")
         (object ClassAttribute "endereço"
         quid          "3D47DFCC0226")
         (object ClassAttribute "cgc"
         quid          "3D47DFCE019A")))
     (object Class "professor"
     quid          "3D47DFE1005A"
     operations    (list Operations
         (object Operation "imprimir"
         quid          "3D47DFFB03B6"
         concurrency   "Sequential"
         opExportControl    "Public"
         uid           0)
         (object Operation "ler"
         quid          "3D598D990208"
         concurrency   "Sequential"
         opExportControl    "Public"
         uid           0))
     class_attributes  (list class_attribute_list
         (object ClassAttribute "nome"
         quid          "3D47DFED037A")
         (object ClassAttribute "disciplina ministrada"
         quid          "3D47DFEF02EE")))
     (object Association "$UNNAMED$0"
     quid          "3D47E00200D2"
     roles         (list role_list
         (object Role "$UNNAMED$1"
         quid          "3D47E0030280"
         supplier      "Logical View::universidade"
         quidu         "3D47DFBD0226"
         client_cardinality     (value cardinality "1")
         is_navigable  TRUE)
         (object Role "estuda em"
         quid          "3D47E0030281"
```

```
    label          "estuda em"
    supplier       "Logical View::aluno"
    quidu          "3D47DF9D00C8"
    client_cardinality     (value cardinality "1..n")
    is_navigable  TRUE)))
(object Association "$UNNAMED$2"
quid           "3D47E00502D0"
roles          (list role_list
    (object Role "$UNNAMED$3"
    quid           "3D47E006017C"
    supplier       "Logical View::professor"
    quidu          "3D47DFE1005A"
    client_cardinality     (value cardinality "1..n")
    is_navigable  TRUE)
    (object Role "$UNNAMED$4"
    quid           "3D47E006017D"
    supplier       "Logical View::aluno"
    quidu          "3D47DF9D00C8"
    client_cardinality     (value cardinality "1..n")
    is_navigable  TRUE)))
(object Association "$UNNAMED$5"
quid           "3D47E0090082"
roles          (list role_list
    (object Role "$UNNAMED$6"
    quid           "3D47E009026C"
    supplier       "Logical View::professor"
    quidu          "3D47DFE1005A"
    client_cardinality     (value cardinality "1")
    is_navigable  TRUE)
    (object Role "$UNNAMED$7"
    quid           "3D47E009026D"
    supplier       "Logical View::universidade"
    quidu          "3D47DFBD0226"
    client_cardinality     (value cardinality "1")
    is_navigable  TRUE)))
(object Class_Category "User Services"
attributes     (list Attribute_Set
    (object Attribute
    tool           "VisualBasic"
    name           "TierPackage"
    value          "User Services"))
quid           "32F8E3FE01E0"
```

A partir deste código gerado pelo Rational Rose, o software desenvolvido para a validação faz a sua leitura e identifica os possíveis erros existentes.

No caso da existência do erro referente a Classe_Com_Métodos, o software informa quais são os métodos presentes e que eles não devem existir. Já no caso de Assiações_Sem_Nome, o software informa quantas são as associações que estão sem nome e que todos as associações devem possuir nomes.

Quando da primeira fase da validação, observou-se em determinados modelos conceituais avaliados a clara existência da necessidade da aplicação das regras como é o caso da figura 31 que demonstra a inexistência de atributos e da figura 32 que apresenta classes sem atributos ligadas a apenas uma outra classe, o que aumenta o grau de certeza da regra.

A partir dessa constatação vê-se a possibilidade da aplicação da regra 4 (Falta_Atributos) com um grande percentual de chances de acerto para o aprendiz.

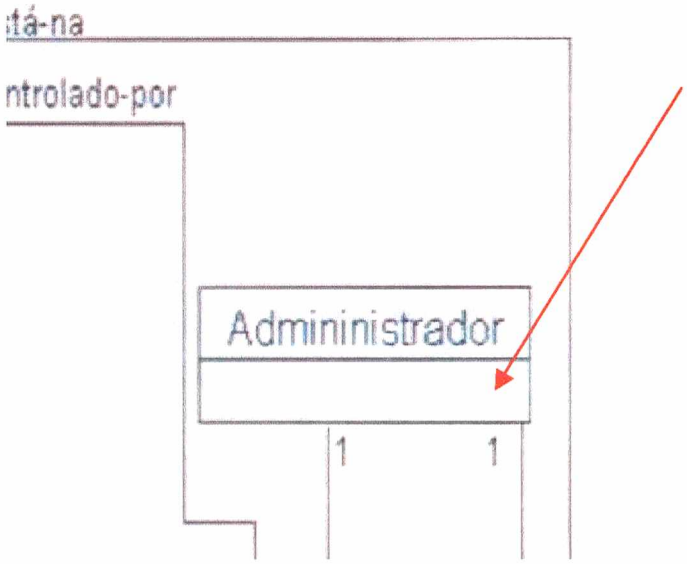


Figura 31 – Inexistência de Atributos

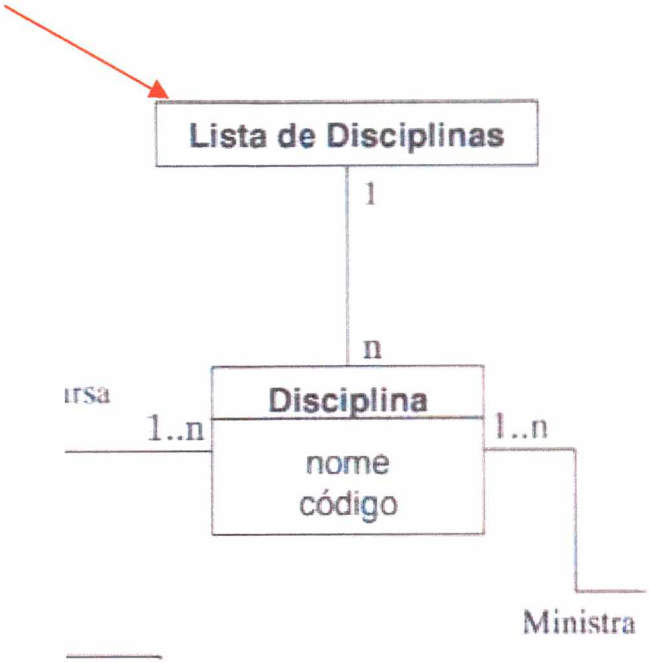


Figura 32 – Classe sem Atributos ligada a uma única Classe

Outro exemplo que pode ser citado está nas figuras 33 e 34 que demonstram inúmeros casos de multiplicidade 1 para 1.

Figura 33 – Multiplicidade 1 para 1

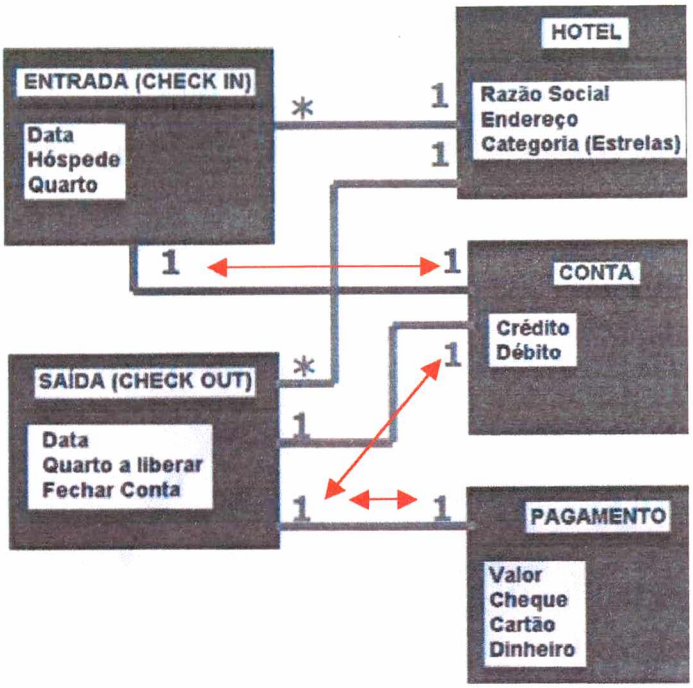


Figura 34 – Multiplicidade de 1 para 1

Neste caso a regra Multiplicidade_Incorreta pode ser aplicada chamando atenção para o fato de que casos de multiplicidade de 1 para 1 são raros.

Um exemplo pode ser associar check Out e conta com multiplicidade de 1 para 1 que é no caso uma relação de 1 para 0..1, então teríamos o erro de modelagem. Já na associação de ckeck In e conta é uma relação de 1 para 1 então não existe erro (neste caso, pode-se sugerir que check In e conta fossem uma só identidade para eliminar a relação de 1 para 1, mas isso não é obrigatório).

Na figura 34, também se observa a não existência de nomes nas associações o que pode ser um erro que deve ser tratado.

Na figura 35 observa-se a presença de atributos repetidos, o que cria a possibilidade da existência de um erro. Há porém, que se observar que nem sempre esta situação representa um erro e que em alguns casos pode ser resolvido com a aplicação de herança.

Outro aspecto a se observar é que quanto mais perto estão as classes onde os atributos estão repetidos, mais aumenta o grau de certeza da regra.

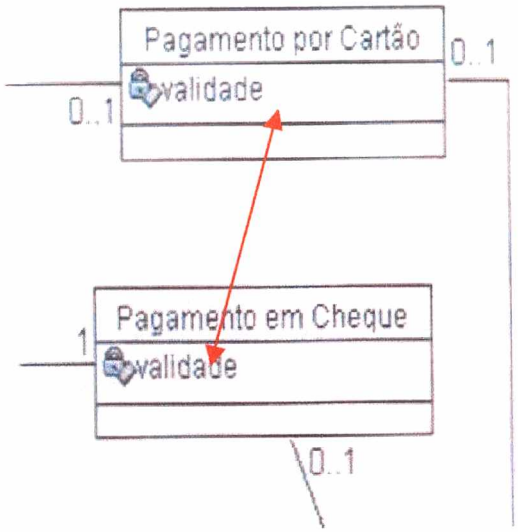


Figura 35 – Atributos Repetidos

Observou-se também o uso de Classes Coletivas, o que implica no erro 2, onde pode-se aplicar a regra 2 (Classes_Coletivas) com alto grau de eficiência. A figura 36 demonstra a existência deste erro no momento da validação da proposta.

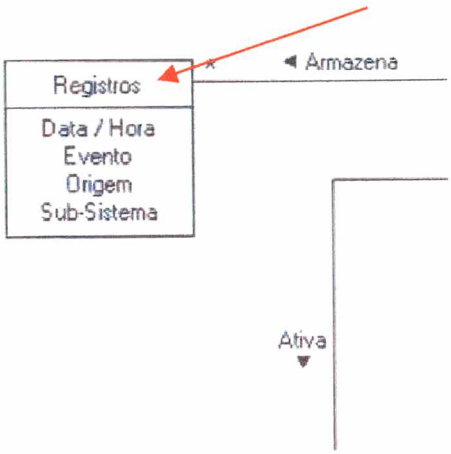


Figura 36 – Modelo com Casse Coletiva

Embora alguns dos erros observados no momento do diagnóstico não foram observados no ato da validação, como é o caso da existência de direcionamentos nas associações e da falta de multiplicidade nas mesmas, por serem estes, erros primários, com certeza, seriam verificados pelo modelo com o maior grau de certeza.

Outros erros, como é o caso do uso de métodos e classes com software, não encontrados podem ser considerados como erros confiáveis, pois sua fonte foi literária e definida por Craig Larman.

.

CAPÍTULO VIII

CONCLUSÕES

Um dos maiores desafios da informática hoje é aliar as facilidades que a Orientação a Objetos proporciona aos percalços que os desenvolvedores baseados neste paradigma encontram em suas experiências, seja em adaptação às novas metodologias ou em conhecimento das ferramentas existentes para tanto.

Conhecendo algumas destas dificuldades, esta pesquisa buscou analisar um dos maiores problemas que os aprendizes deste paradigma encontram no momento do desenvolvimento da modelagem conceitual.

Para que erros fossem evitados é que se procurou desenvolver a proposta de tratamento para os mesmos, o que foi desenvolvido baseado nos trabalhos realizados por alunos de graduação e pós-graduação.

A partir da identificação, os erros mais comumente encontrados, isto em quantidade de trabalhos analisados é que foram propostas as regras e os algoritmos necessários para o desenvolvimento dos mesmos e também a implementação do software desenvolvido para a validação.

Para que estas regras fossem definidas, foi utilizada a notação BNF Estendida, que formalizou os dados referentes à pesquisa e pode-se, através destes dados, também desenvolver funções booleanas que auxiliassem no desenvolvimento dos algoritmos necessários para que a validação do mesmo fosse possível. A maioria destas funções não foi desenvolvida ou especificada formalmente, ficando para um trabalho futuro.

Com o desenvolvimento destas regras aplicadas a uma ferramenta CASE, muitos dos erros efetuados na modelagem conceitual podem ser evitados desde que recebam o devido tratamento, e que este tratamento seja implementado de forma eficaz.

A relevância do trabalho se dá pelo fato de que muitos alunos erram acreditando estar desenvolvendo modelos conceituais consistentes e, só percebem suas falhas em fases posteriores onde o erro já está atrapalhando o desenvolvimento final do software.

Com a definição destas regras torna-se possível para o aluno, antes mesmo de passar para a fase seguinte do projeto identificar esses erros e possivelmente fazer a correção dos mesmos, assim tornando o projeto mais consistente.

8.1. Trabalhos Futuros

Durante todo o Capítulo V foram levantados textualmente alguns problemas que ainda ficaram obscuros para a pesquisa, sendo que os mesmos podem ser classificados como:

a) No erro 4 que é Falta_Atributos: Se a classe estiver ligada a mais de uma outra classe, como detectar se ela é útil como elemento de associação ou não?

b) No erro 5, Atributos_Repetidos: Como tratar os atributos que realmente se repetem em diferentes classes? Como identificar se houve realmente redundância?

c) No erro 6, Associações_Incorretas: Como identificar um os verbos que denotam transformações?

Além desses pontos que ficaram em aberto no modelo proposto também é importante ressaltar que além destes erros ainda podem existir outros, que se identificados podem ser incorporados ao modelo.

A análise e o diagnóstico desenvolvidos nesta dissertação de mestrado abre a possibilidade de desenvolvimento de uma ferramenta CASE que seria responsável por "avisar" os aprendizes na hora em que os erros fossem identificados para que os mesmos revissem suas definições, o que traria vantagens, pois, ao identificar o erro, este já poderia ser solucionado.

REFERÊNCIAS

(ATKINSON, 1989) ATKINSON, M. et alii – **The Object Oriented Database System Manifesto Proceedings of Object Oriented Database Management System** – Kyoto – Japan – December, 1998.

(BARBIERI, 1997) BARBIERI, C. – **O Método de Análise Orientada a Objetos** – Unified Modelinf Language, 1997.

(BJR, 1997) BOOCh J. et alii – The UML **Especification Document** – Santa Clara – Califórnia – Rational Software Corporation – Disponível na URL http://www.rational.com

(BUZATO, 1998) BUZATO, LUIZ E., et alii. – **Construção de Sistemas Orientados a Objetos Confiáveis** – 11ª escola de Computação – Julho de 1998.

(BOOCH, 1994) BOOCH, G – **Object Oriented DesignWith Applications** – Benjamin Armmings - Redwood – 1991.

(COAD, 1995) COAD P. – **Object Models: Strategies, Patterns and Applications** – Englewood Cliffs – NJ – Pretence Hall – 1995.

(COLEMAN, 1994) COLEMAN, D. et alii, **Object-Oriented development: The Fusion Method** – Englewood Cliffs, NJ: Pretence-Hall – 1994.

(FOWLER, 1996) FOWLER, M. - **Analysis Patterns: Reusable Objet Models reading**, MA: Addison-Wesley, 1996.

(FURLAN, 1998) FURLAN, José Davi – **Modelagem de Objetos através da UML – The Unified Modeling Language** – São Paulo – Makron Books – 1998.

(JACOBSON, 1992) Disponível em http://rational.com.br, em 20/07/2002.

(JAIME, 2002) Disponível em http://www.gold.com.br/~jaime/textos/uml.htm em 20/07/2002.

(KNUDSEN, 1994) KNUDSEN, J. et alii – **Conceptual Framework. In Object Oriented Environment** – Cambridge University, Chapter 4 – 1994.

(LARMAN, 2000) LARMAN, C. – **Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos** – Trad. Luiz A. Meirelles Salgado – Porto Alegre – Bookman – 2000.

(LARMAN, 2002) LARMAN, C. –**Applying UML ang Patterns – as Introduction to Object-Oriented Analysis and the Unified Process** - Second Edition. Pretence hall: Upper Saggle River, 2002

(LEITE, 2001) LEITE, Jair C. **Notas de Aula** – Disponível na URL http://www.dimap.ufrn.br/~jair/ES/c5.htm#5.2 em 12/092001

(MARCELLO, 2002) Disponível em http://www.dct.ufms.br/~marcelo/teaching/too/texts/aula11.pdf em 20/07/2002.

(MARTÍN, 2002) Disponível em http://rau-tu.fai.com.br/sd/answer.php?qid=1 em 20/07/2002.

(MARINHO, 2001) MARINHO, Fabiana Gomes – **Padrões para Integração de Visões Modeladas com UML** – Dissertação de Mestrado – Universidade Federal do Ceará – 2001.

(MATTOS, 1999) MATTOS, M. – **Emprego de Métricas de Qualidade em um Protótipo de uma Ferramenta Case para a UML** – VIII SEMINCO – Seminário de Computação – 1999.

(MONARD, 1992) MONARD, Maria Carolina et alii – **Uso de Programação Lógica no Desenvolvimento de Compiladores** – Universidade de São Paulo/ILTC – Disponível na URL

http://labic.icmsc.sc.usp.br/didatico/PostScript/compiladores.html em 04/01/2002.

(MULLER, 1997) MULLER, Piere Alain – **Instant UML** – Paris – Editions Eyrolles – 1997.

(MUNIZ, 2001) MUNIZ, Leonel – **Orientação a Objetos** – Disponível na URL http://www.angelfire.com/ok/leonelmuniz/p13.html - em 17/09/2001

(NETO, 1987) NETO, José et alii – **Introdução a Compilação** – Livros Técnicos e Científicos, 1987 - Disponível na URL http://www.hipernet.ufsc.br/foruns/Convidados/Computac.../compiladores2.htm em 04/01/2002.

(NICOLAS, 2000) Disponível na URL http://www.cos.ufrj.br/~nicolas/UML em 17/12/2001.

(NUNES, 2001) Disponível na URL http://icmsc.sc.usp.br/~mdgvnune/LP/fp.htm em 17/12/2001.

(OMG, 1994) Object Management Group: Object Models. **OMG Document** 94-10-10, October, 1994.

(PALAZZO, 2002) Disponível em http://www.inf.ufrgs.br/~gameiro/ti_reuso.htm, em 20/07/2002.

(PEREZ, 2000) PEREZ, Andres et alii – **FEMA Virtual** – Disponível na URL http://www.femanet.com.br/fema_virtual/analise/html/ em 25/10/2001.

(RATIONAL,2000) RATIONAL, Software corporation – **Unified Modeling Language** – Versão 1998 – Disponível na URL http://www.rational.com/uml em 05/06/2001.

(RUMBAUGH, 1991) RUMBAUGH J. et alii – **Object Oriented Modeling and Design** – Pretence Hall – Englewood Cliffs – 1991.

(RUMBAUGH, 1994) RUMBAUGH, J. et alii – **Modelagem e Projetos Baseados em Objetos** – Rio de Janeiro – Editora Campus – 1994.

(SEBESTA, 2000) SEBESTA, Robert W. **Conceitos de Linguagens de Programação**. 4ª ed. Porto Alegre: Bookman - 2000.

(ZANCANARO, 1999) ZANCANARO, A. et alii – **Utilização da UML para a Especificação de Sistemas de Tempo Real** – VIII SEMINCO – Seminário de Computação – 1999.

ANEXO 01

Baseado na formalização dos dados (Capítulo VI) definidos como necessários para o desenvolvimento da idéia algorítmica, foram desenvolvidas regras que poderão vir a ajudar numa possível implementação de melhoria das ferramentas CASE existentes e das ferramentas futuramente desenvolvidas para a modelagem dos dados de um projeto computacional baseado em Orientação a Objetos.

1. Regra 1: Classe_Plural

Varre a lista de conceitos e verifica se o nome da classe termina com S, retornando T (True) ou F (False) para a função definida anteriormente. Se retornar T "True" significa que o nome da classe termina em S, então está no plural. Se retornar F "False" a classe não está no plural.

Início
Definir:
(Modelo:<Modelo>)
(Conceito:<Conceito>)
Plural (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Conceito* e
Se elemento é *NomeClasse* e
Se Plural (nome(elemento)) ou
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Classe_Plural!!!");
Escreva ("Conjunto de elementos são denotados na multiplicidade das associações e que não se criam classes para denotar estes conjuntos, a não ser que estes conjuntos possuam atributos próprios que não podem ser derivados dos atributos de seus elementos.");;
Fim Então;
Fim Se;
Fim Se;

Fim Se;

Fim Se;

Fim Para;

Fim.

2. Regra 2: Classe_Coletiva

Varre a lista de conceitos e verifica se o nome da classe é um substantivo coletivo, retornando T (True) ou F (False) para a função definida anteriormente. Se retornar T "True" significa que o nome da classe termina em S, então está no plural. Se retornar F "False" a classe não está no plural.

Início
Definir:
(Modelo:<Modelo>)
(Conceito:<Conceito>)
Plural (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Conceito* e
Se elemento é *NomeClasse* e
Se Coletivo (nome(elemento))
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Classe_Coletiva!!!");
Escreva ("Conjunto de elementos são denotados na multiplicidade das associações e que não se criam classes para denotar estes conjuntos, a não ser que estes conjuntos possuam atributos próprios que não podem ser derivados dos atributos de seus elementos.");;

Fim Então;

Fim Se;

Fim Se;

Fim Se;

Fim Se;

Fim Para;

Fim.

3. Regra 3: Direcionamentos_Incorretos

Varre toda a lista de associações e verifica se foram colocados. Se retornar T "True" foram colocados direcionamentos nas associações. Se retornar F "False" está correto (sem direcionamento).

Início
Definir:
(Modelo:<Modelo>)
(Associação:<Associação>)
(Direção:<Direção>)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Associação* e
Se elemento é *Direção*
Se *Direção* ← "→"| "←"| "↔"
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Direcionamentos_Incorretos!!!");
Escreva ("Não se colocam direcionamentos no Modelo Conceitual. Somente com as informações dos Diagramas de Colaboração da fase de projeto é que se poderá saber como direcionar as associações de maneira apropriada.");
Fim Então;
Fim Se;
Fim Se;
Fim Se;
Fim Para;
Fim.

4. Regra 4: Falta_Atributos

Varre toda a lista de conceitos e informa a existência de classes sem atributos ou com apenas um atributo. Retorna T "True" se for verdadeiro e F "False" caso contrário.

Início
Definir:

(Modelo:<Modelo>)
(Atributo:<Atributo>)
Trivial (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Atributo* e
Se elemento é *NomeAtributo* e
Se Trivial (nome(elemento))
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Falta_Atributos!!!");
Escreva ("Classes sem atributos ou com apenas um único atributo devem ser observadas com atenção, pois se não possuir nenhum atributo ela talvez possa ser dispensada e esta classe pode ser implementada como um atributo simbólico em outra classe a que ela faz referência.");
Fim Então;
Fim Se;
Fim Se;
Fim Se;
Fim Para;
Fim.

5. Regra 5: Atributos_Repetidos

Varre todo o modelo conceitual, identificando a existência de atributos repetidos em duas ou mais classes. Se estes existirem a função retorna T "True" e se não existirem retorna F " False".

Início
Definir:
(Modelo:<Modelo>)
(Atributo:<Atributo>)
Compara (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Atributo* e
Se elemento é *NomeAtributo* e
Ler *NomeAtributo*
Escrever *NomeAtributo* //Coloca todos os nomes dos atributos em um Vetor V
Fim Se;
Fim Se;
Fim Para;

Para Todo elemento **de** V **Faça**

Se Compara (nome(elemento))

Então

Escreva ("Foi detectado um possível erro de modelagem !!! Atributos_Repetidos!!!");

Escreva ("Os atributos que se encontram repetidos em várias classes podem estar sendo repetidos desnecessariamente, talvez, uma associação entre duas classes resolva este problema")

Fim Então;

Fim Se;

Fim Para;

Fim.

6. Regra 6: Associações_Incorretas

Verifica o nome de todas as associações informando que certos verbos não podem ser utilizados por denotarem transformações de informações. A função Verbo definida anteriormente facilita a resolução do algoritmo.

Início

Definir:
(Modelo:<Modelo>)
(Associação:<Associação>)
VerboDeTranformação (x)

Para Todo elemento **de** *Modelo* **Faça**

Se elemento ∈ *Associação* e

Se elemento é *NomeAssociação* e

Se VerboDeTransformação (nome(elemento))

Então

Escreva ("Foi detectado um possível erro na modelagem!!! Associações_Incorretas!!!");

Escreva ("Associações devem representar informações estáticas sobre a estrutura de conceitos e não as transformações dinâmicas");

Fim Se;

Fim Se;

Fim Se;

Fim Para;

Fim.

7. Regra 7: Atributo_Que_É_Referência

Varre a lista de atributos e verifica se algum de seus nomes é referência para outra classe, além de verificar se algum atributo possui o nome de alguma classe do mesmo modelo conceitual. As funções definidas anteriormente informam a existência deste possível erro, dizendo que se deve usar associações para unir conceitos e não atributos.

Início
Definir:
(Modelo:<Modelo>)
(Atributo:<Atributo>)
Referência (x)
Compara (x)
Para Todo elemento *de Modelo* **Faça**
Se elemento ∈ *Atributo* e
Se elemento é *NomeAtributo* e
Se Referência (nome(elemento))
Então
Escreva ("Foi detectado um possível erro na modelagem!!! Atributo_Que_É_Referência!!!");
Escreva ("Usa-se associações para ligar conceitos e não atributos.");
Senão;
Para Todo elemento **de** Modelo **Faça**
Se elemento é Conceito e
Ler *NomeClasse*
Escrever *NomeClasse*
Ler *NomeAtributo*
Escrever *NomeAtributo*
Fim Se;
Fim Para;
// Todos os nomes são escritos em um vetor que foi nomeado por V
Para Todo elemento **de** *V* **Faça**
Se Compara (nome(elemento))
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Atributos_Incorretos!!!");
Escreva ("Os atributos não podem possui o mesmo nome que as classes")
Fim Então;
Fim Se;
Fim Para;
Fim Senão;

Fim Se;

Fim Se;

Fim Se;

Fim Para;

Fim.

8. Regra 8: Sem_Multiplicidade

Verifica a multiplicidade em ambos os lados da associação e, se não houver nenhum sinal de multiplicidade, informar que pode estar acontecendo um erro e que pode ser aplicada a regra existente para este caso.

Início
Definir:
(Modelo:<Modelo>)
(Multiplicidade:<Multiplicidade>)
(Associação:<Associação>)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Associação* e
Se elemento é *Multiplicidade* e
Se *Multiplicidade* → ε
Então
Escreva ("Foi detectado um possível erro na modelagem!!! Sem_Multiplicadade!!!");
Escreva ("Associações sem anotação de multiplicidade são consideradas como sendo de 1 para 1.");

Fim Então;

Fim Se;

Fim Se;

Fim Se;

Fim Para;

Fim.

9. Regra 9: NomeAssociação_Incorreto

Verifica o nome de todas as associações informando que associações do tipo "é"ou "é-um" implica no Erro 9 e deve ser aplicada a Regra que fala sobre os Nomes das Associações Incorretos.

Início
Definir:
(Modelo:<Modelo>)
(Associação:<Associação>)
Herança (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Associação* e
Se elemento é *NomeAssociação* e
Se Herança (nome(elemento))
Então
Escreva("Foi detectado um possível erro de modelagem!!!NomeAssociação_Incorreto!!!");
Escreva ("Trata provavelmente de herança, ou que o nome da associação é inadequada.");
Fim Então;
Fim Se;
Fim Se;
Fim Se;
Fim Para;
Fim.

10. Regra 10: Multiplicidade_Incorreta

Verifica a multiplicidade em ambos os lados da associação e se o sinal de multiplicidade for 1 em ambos os casos, informar que são raros os casos de em que este tipo de multiplicidade ocorre.

Início
Definir:

(Modelo:<Modelo>)
(Multiplicidade:<Multiplicidade>)
(Associação:<Associação>)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Associação* e
Se elemento é *Multiplicidade* e
Se *Multiplicidade* = "1"
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Multiplicadade_Incorreta!!!");
Escreva ("Associações com multiplicidade 1 para 1 são raras de acontecer, observe novamente se não houve algum esquecimento.")
Fim Então;
Fim Se;
Fim Se;
Fim Se;
Fim Para;
Fim.

11. Regra 11: Classes_Com_Métodos

Verifica se as classes possuem métodos, o que no modelo conceitual não é permitido, ficando eles para os Diagramas de Colaboração, informa-se neste caso o erro e a possibilidade de aplicação da regra que trata das classes possuidoras de métodos.

Início
Definir:
(Modelo:<Modelo>)
(Conceito:<Conceito>)
Método (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Conceito* e
Se elemento é *Método* e
Se Método (nome(elemento))
Então
Escreva ("Foi detectado possível erro de modelagem!!! Classes_Com_Métodos!!!");
Escreva ("Não se usa métodos no modelo conceitual, por se tratar de entidades de software e não de elementos do mundo real")
Fim Então;
Fim Se;

Fim Se;

Fim Se;

Fim Para;

Fim.

12. Regra 12: Atributos_Com_Códigos

Varre a lista de atributos e verifica se algum de seus nomes apresenta o nome de "código" ou semelhante, informa que somente códigos do mundo real são aceitos no modelo conceitual. Ex: CPF, CGC, etc.

Início
Definir:
(Modelo:<Modelo>)
(Atributo:<Atributo>)
Código (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Atributo* e
Se elemento **é NomeAtributo** e
Se Código (nome(elemento))
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Atributos_Com_Códigos!!!");
Escreva ("Usa-se somente códigos do mundo real no modelo conceitual, Ex: CPF, CGC, etc.");
Fim Então;
Fim Se;
Fim Se;
Fim Se;
Fim Para;
Fim.

13. Regra 13: Classes_Com_Software

Varre a lista de classes e identifica nomes referentes a componentes de software. Indica erro e possibilidade da aplicação da regra que trata deste caso.

Início
Definir:
(Modelo:<Modelo>)
(Conceito:<Conceito>)
Software (x)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Conceito* e
Se elemento é *NomeClasse* e
Se Software (nome(elemento))
Então
Escreva ("Foi detectado um possível erro de modelagem!!! Classes_Com_Software!!!")
Escreva ("O Modelo Conceitual trata de elementos do mundo real e não de componentes de software.
Fim Se;
Fim Se;
Fim Se;
Fim Para;
Fim.

14. Regra 14: Associações_Sem_Nome

Varre a lista de classes e identifica nomes referentes a componentes de software. Indica erro e possibilidade da aplicação da regra que trata deste caso.

Início
Definir:
(Modelo:<Modelo>)
(Associação:<Associação>)
Para Todo elemento **de** *Modelo* **Faça**
Se elemento ∈ *Associação* e
Se elemento é *NomeAssociação* e
Se *NomeAssociação* = ε
Então
Escreva ("Foi detectado um possível erro de modelagem!!!Associações_Sem_Nome!!!");
Escreva ("Todas as Associações devem possuir nomes");
Fim Então;
Fim Se;

Fim Se;

Fim Se;

Fim Para;

Fim.

A definição de regras semânticas, ou seja, a maneira como o programador irá desenvolver seu software não faz parte do escopo deste trabalho.

Os algoritmos foram desenvolvidos em uma linguagem algorítmica híbrida, ou seja, foram utilizadas várias técnicas algorítmicas para tanto.

ANEXO 02

```
#include <stdio.h>
int associacao=0;
main()
{
 FILE *P;
 int x=0;
 char ch, string[80];
 clrscr();
 printf("\n\n  INFORME O CAMINHO E O NOME CORRETO DO ARQUIVO A SER ANALISADO\n\r
Ex: c:\\temp\\arquivo.txt ==> ");
 gets(string);
 P=fopen(string,"rb");
 if(P==NULL)
        {
                printf("\n ERRO NA ABERTURA DO ARQUIVO VERIFIQUE O CAMINHO E O NOME
CORRETO DO MESMO.");
                fclose(P);
                getch();
                exit(0);
        }
 while(!feof(P))
        {
                        ch=fgetc(P);
                        string[x]=ch;
                        string[x+1]='\0';
                        if(ch=='(' || ch=='d')
                        {
                                x=0;
                                string[0]=ch;
                        }
                        x++;
                        if (strcmp(string,"(object Class ")==0)
                        {
                                while(!feof(P))
                                {
                                        ch=fgetc(P);
                                        string[x]=ch;
                                        string[x+1]='\0';
                                        if(ch=='(' || ch=='d' || ch=='s' )
                                        {
                                                x=0;
                                                string[0]=ch;
                                        }
                                        x++;
                                        if (strcmp(string,"documentation")==0)
                                        {
```

```
                cprintf("\n\n        ATENÇÃO!!! FOI ENCONTRADO %d ASSOCIAÇÔES SEM NOME. ISTO É UM POSSIVEL ERRO DE MODELAGEM, POIS TODAS AS ASSOCIAÇÕES DEVEM POSSUIR UM NOME.\n\r
Pressione enter para continuar ==> ",associacao);
                getche();
                cprintf("\n\n  FINAL DA ANÁLISE DO ARQUIVO. \n\r
Pressione enter para continuar ==> ");
                fclose(P);
                getch();
                exit(0);
        }
        if (strcmp(string,"(object Operation \"")==0)
        {
                x=0;
                while(!feof(P))
                {
                        ch=fgetc(P);
                        string[x]=ch;
                        string[x+1]='\0';
                        if(ch=='"')
                        {
                                string[x]='\0';
                                cprintf("\n\n        ATENÇÃO!!! FOI DETECTADO UM POSSIVEL ERRO DR MODELAGEM. A OPERAÇÃO \"%s\" É ILEGAL, POIS REPRESENTA ENTIDADE DE SOFTWARE E NOS MODELOS CONCEITUAIS SOMENTE SÃO ACEITOS ELEMENTOS DO MUNDO REA.\n\r  Pressione enter para continuar ==> ",string);
                                getche();
                                break;
                        }
                        x++;

                }

        }
        if (strcmp(string,"s_navigable \tTRUE)\r")==0)
        {
                x=0;
                while(!feof(P))
                {
                        ch=fgetc(P);
                        string[x]=ch;
                        string[x+1]='\0';
                        if(ch=='(')
                        {
                                x=0;
                                string[x]=ch;
                        }
                        if(strcmp(string,"(object Role \"$UNNAMED")==0)
                        {
                                associacao++;
                                string[x]='\0';
                                break;
                        }
                        x++;
```

```
                                        if(x > 25)
                                                break;

                                    }

                                }
                            }
                        }
                    }
                }
```